1

[redacted]

TM-5897/000/00
30 June 1977

USER LANGUAGE SPECIFICATION
TECHNICAL REPORT (CDRL A006)

SPONSORED BY: Defense Advanced Research Projects
Agency (DOD)
ARPA Order No. 3037, Amendment 12

MONITORED BY: NAVELEXSYSENGCEN, POW-1, Vallejo
Under Contract No. N00228-77-C-2070
Contract Dates: 3 December 1966 to
30 June 1977
Reporting Period: 30 June 1977

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

500 Macara Avenue • Sunnyvale, California 94086 • (408) 737-8000

TM-5897/000/00
30 June 1977

USER LANGUAGE SPECIFICATION
TECHNICAL REPORT (CDRL A006)

PRINCIPAL AUTHORS

PAUL R. EGGERT (213) 825-7511, X3473
ROBERT C. UZGALIS (213) 825-7511, X3473

APPROVED BY

SDC/ARC PROGRAM MANAGER, DR. E. LEVIN

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

The task objective that guided the work reported in this document was to develop the ARC User Language Specification based on user requirements, candidate approaches, and trade-off analyses. This work was motivated by the need to achieve easier interaction between experimenters and the ARC in order to decrease the time required to develop an experiment. The work resulted in a specification of the Network Language LINGO which is included as Appendix A of this document.

Technical problems considered in the development of LINGO included: the need for evolutionary implementation of the design without obstructing use of the ARC for ongoing experiments, system synchronization problems including prevention of deadlock, performance measurement and prediction, detecting errors in hooking programs together, the desirability of adapting existing software rather than replacing it, and the distributed system characteristics of both present and future hardware configurations.

The general methodology consisted of a four-step approach:

1. Exploration of the problem and relevant previous work, including the examination of previous ARC experiments, evaluation of existing ARC software, and conversations with ARC users and members of the Systems Applications Branch to determine requirements.
2. Formulation of the general structure of a solution, in which many alternatives were proposed and discarded on the basis of requirements, or logical or engineering flaws. This resulted in the selection of the link-node, data-driven model on which LINGO is based.
3. Test and refinement of the link-node model by applying it to more detailed aspects of the requirements and proposed METAEXEC architectures.
4. Completion of the formal specification of the LINGO Language which involved many language changes to guarantee logical consistency.

The technical results of the LINGO design activity consist of a formal specification of the syntax, an informal semantics of the experiment design portion of LINGO, and specification of the user/equipment interface and computer programs required to support use of the language. These results are presented in Sections 2 through 4 and in Appendix A of this report. The most significant characteristic of LINGO is the degree to which it contributes to discovery and prevention of system synchronization and resource allocation problems before operation of an experiment. The rationale for the related design decisions is explained in Section 2.

The important conclusions resulting from this work are:

1. System synchronization problems can be eliminated by the appropriate design of an experiment design language, its language processor and a meta-executive. This type of disciplined solution is needed because of the extreme difficulty of testing and debugging synchronizing errors when ad hoc solutions are used.
2. Performance prediction for an experiment network can be supported by an experiment design language and its processor, subject to certain conditions and limitations. Accurate manual estimates of the performance of primitive nodes and links will be required, and additional work is required on the derivation of the mathematical relationship between primitive node performance and network performance.
3. Previous recommendations for phased development are supported by the results of this work.

As a result of the work on LINGO and reactions to the Design Review of June 7-8, the following implications for further work have been identified:

1. The command language, for experimenter use in controlling experiment operation, requires further design work. This design should be closely coordinated with LINGO and the design of the METAEXEC system software.
2. Design of system nodes and common signal processing nodes, plus more detailed design of the METAEXEC system software is needed to provide inputs for refinement of LINGO, as well as for their primary purposes. The concern about adequate flexibility resulting from the Design Review of June 7-8 indicates the need for a deeper understanding of the capabilities of the design and how to exploit those capabilities in relation to the flexibility needed by experimenters.
3. Development of the mathematical basis for prediction of the performance of a LINGO experiment network is recommended. LINGO (see Section 2 and Appendix A) provides a notation for including

within a processing node specification of the primary information needed for performance prediction. However, the algorithms for calculating network performance from the primary information and the degree of precision feasible have not yet been devised.

4. LINGO user's manuals will be needed. Two audiences exist, experiment designers and LINGO library designers.

5. Language processors will be needed. One processor, called the LINGO linker, should be produced to analyze an experiment network, link its nodes, and produce the control tables required for operation of the experiment. A second processor should be produced to interpret commands during operation of an experiment. It is expected that additional utility processors will be needed, such as experiment loaders and node library managers, but their functions and position in the structure of the system have not yet been determined.

1.0 INTRODUCTION

This document is a report of design work on the user language and computer programs required to support it. LINGO is divided into two major parts, experiment description and experiment control. Because of dependency relationships among these major language parts and the system support software, the experiment description portion of LINGO has been designed and is presented here. Design of the experiment control portion of the user language remains to be done.

Sections 1.1 through 1.4 provide background information, including a survey of relevant previous work and a description of the language design criteria derived from the requirements. Section 2 contains a general description of LINGO with explanations of reasons for the critical design decisions. A complete formal specification of the LINGO language is in Appendix A. Section 3 contains a tentative specification of the user equipment interface. Section 4 describes in general terms the computer programs required to support the LINGO language. Section 5 is the test and verification specification for the language.

1.1 SPECIFYING COMPUTATIONAL TASKS

The development of a method for reliably specifying large scale computational tasks by technically competent people has existed since Fortran attempted to provide a solution in the mid 1950s. By-and-large, Fortran was a resounding success in providing computer access to non-programmers. However, the problems that were being solved in the mid-50s are miniscule compared to the problems that the computer is being used to attack in the mid-70s. Fortran is no longer an adequate solution because the problems have changed and increased in complexity. The computation of a function is rarely the problem today, rather the design of appropriate functions and a reliable way of interconnecting related functions to perform a large integrated task are the new frontier.

Top-down design and structured programming are directed toward the identification and implementation of appropriate functions for the performance of a given task. This language design effort is directed toward the problem of interconnecting those functional modules in a reliable way. LINGO is specifically aimed at large-scale computational tasks that do not require the construction of new functions but rather employ existing well defined and reliable programs.

LINGO has been specifically constructed to look forward to processing problems of the future. It allows implementation on multi-machine configurations as well as uni-processors. It encourages overlapped processing of functions. It allows some degree of performance prediction indispensable in real-time processing and control.

1.2 SURVEY OF INTERCONNECTION TECHNOLOGY

Jack Dennis of MIT has been interested in the problems of interconnecting functions for many years. He has written several technical reports that serve to outline the subtle problems which characterize the interconnection of modules and posed possible solutions [5]. Much of what follows has been drawn from his insight and understanding.

The interconnection of independent modules has been done since before Fortran. The first Fortran facility on the IBM 702 provided automatic library resolution of the Fortran-defined functions and run-time routines, although user-defined functions had to be compiled individually and were then automatically linked. Soon user libraries of functions and subroutines appeared. At this point the real troubles began. Mismatch of the type or meaning of the argument to the corresponding parameter made libraries somewhat unreliable to use and generally pretty limited because the functions available were too specific. For efficiency, many systems passed Fortran function data in common, creating many more problems than were solved using the technique and even more narrowly limiting the use of the subroutine library. Even now initialization and proper linkage of independent Fortran subroutines are difficult and error prone.

To improve over the situation that obtained in Fortran, PL/I introduced the EXTERNAL attribute and solved some of the problems. The problem of a consistent definition of the external data was attacked by allowing the inclusion of source text from a library by the compiler. To guarantee consistent use the data description was placed in a library and "included" in each routine where it was used. If the description of the common data was to be modified, then the library copy was modified and each independent subroutine which used the data was recompiled.

The designers of Algol68C, a compiler for Algol 68 on the IBM 360/370 series of machines, took an alternative approach. Instead of requiring the user to provide multiple linked declarations, the compiler maintains and "environment file" which retains all declarative information. Programs must be compiled in the appropriate order so that the environment already exists within the file, otherwise an error is detected. This solves most of the bad linkage problems, but does not solve the general library module problem because it assumes and builds one large program even though elements are separately compiled and modifiable.

The need for a data sensitive linkage editor has been expressed in conjunction with a somewhat simplistic formulation by R. G. Hamlet in the Communications of the ACM [7]. This solution would check the type of parameter and argument and link only when they matched in all cases. However, when structures are considered, the equivalence of two data descriptions becomes much more troublesome and a solution that worked for one language might not work for another. In this situation inter-language linking becomes a nightmare. If types are considered to be more general entities than just a description of the machine format, the linker must become even more sophisticated.

Alternatively a language can be specified that is designed to aid in interconnecting independently built modules. There are several languages that treat other problems in this way, including Gardner [6] and others.

One promising approach toward realistic interconnection of modules is offered by a data flow graph. The source of the data is indicated followed by a series of blocks that represent processing. Running from each processing block or node are links that contain processed data. These links connect to another node and become the source of data for further processing.

This link-node technique seems like a fairly natural method for conceiving of large scale programs. Several people have investigated languages and formal systems for specifying general computation in graph form. The most well known are the graph model of computation of Estrin and Martin [11], Petri nets, and the data flow language/machine of Jack Dennis [5]. These systems are all aimed at computing any function and are capable of simulating other machines. In this sense they are data-oriented analogues of the algorithmic-oriented Turing machine.

1.3 DATA FLOW LANGUAGES

Data flow languages bring to the front a different series of problems than do algorithmic languages. The description of the data flow is inherently parallel and thus subject to race conditions and deadlock. Differences in processing and communication rates allow queues to build up on interconnecting links. Data that is time sampled and continuously flowing creates synchronization and control problems.

It should be a property of a reasonable data flow based network language that these problems can be handled in a system-independent way by only examining the network and a finite set of node performance properties. In advance of executing a network it should be known that it will perform consistently and not unexpectedly terminate (deadlock), loop, or produce unreliable answers due to race conditions. The meta-executive that controls such a network should schedule node execution in such a way that network processing is never terminated because queues overflow on a network link.

To allow programming-in-the-large by technically competent but non-programming people requires that the language behave in a reliable way that generates diagnostics in terms that are understandable to a user, not just to a systems programmer or implementer of a processing node. A non-programmer will, in general, be unaware of the amount of computing resources necessary to perform a given function. It is important, therefore, to be able to reflect network computing costs in whole and on a part by part basis to the user. These costs are in terms of computer cycles and memory because there is no real requirement for dollars and cents estimates.

Since the user has not constructed the processing nodes himself, the interconnections should be checked in advance of execution of the network. At the very minimum, the machine format of data should be checked for compatibility and even the conceptual nature of the data could be checked by including it within the data type. This was proposed earlier by Meertens [12] and also incorporated into algorithmic languages by Cleaveland [4].

It is unrealistic to expect the unsophisticated user to check for race conditions and deadlock; therefore, the language processor must be capable of automatically checking a network in advance of execution. These were the goals that motivated the development of the Network Language LINGO.

1.4 THE ARC ENVIRONMENT

The ARPA-ARC is an environment in which a network data flow language is especially applicable. The ARC provides a network of specialized and general purpose computers that allow higher performance analysis of time series multichannel acoustical data. The primary users of the ARC are scientists with some computer background, but they are really specialists in their own fields of signal processing and ocean physics. The current ARC configuration requires fairly intimate knowledge of both its hardware and software characteristics for effective use to be made of it. The production of an experiment tends to be costly and time consuming, limiting the speed with which research can be conducted.

A network language in this environment would allow users to build their own experiments rather than have to rely on programmers to build the experiments for them. Specification of experiment topology done in LINGO allows easier communication of processing concepts for everyone involved.

Viewed from the point of view of a research person, the ARC is a facility that has a variety of special functions that operate on a common stream of experimental data. The ARC system functions as the preprocessor of data for a researcher's program. For most users of the ARC, it is a specialized machine for doing signal preprocessing. For these users, the data flow from transducer to program is the most important single consideration.

Other considerations include notational overhead imposed on a researcher to use the ARC system, and the amount of cognizance of other researchers' activities required in order to make resource allocation reasonable.

From the point of view of ARC administration, two facets are important. ARC researchers come and go and the problems/research they do change. Therefore, it should be easy for a new researcher to learn to use the ARC resources. Detailed checking coupled with good diagnostic messages will help teach a new user. It will also keep more knowledgeable researchers from making simple errors that might result in unnecessary and costly runs. Secondly, the hardware available is subject to change and, therefore, the researcher should be relatively insensitive to the particular hardware configuration.

The personnel within the ARC-configuration can be categorized into conceptual roles which people can be expected to play at one time or another. These roles do not represent real people; in fact, one person may play several roles or a group may act in one role. The use of the categorization is to allow conception of the support functions played by individual pieces of software.

For these purposes, five fundamental roles are identified:

1. Director: can either be a single person or a committee of principal investigators who have the responsibility to allocate resources during a real-time experiment (i.e., when allocation is too important to be left to algorithms).
2. Principal Investigator: the person in charge of running an experiment. He will probably have a programmer or two responsible for building specific modules necessary for his particular needs.
3. Programmer: the person responsible for building specific experiment-oriented modules in Fortran, Assembler, etc.
4. Node Library Designer/Programmer: a person who will probably be part of the SDC System Support Staff and who has the responsibility to design and build general modules for the Principal Investigators and Programmers to use.
5. Systems Programmer: a person who is part of the SDC System Support Staff having the maintenance responsibilities for the language processors, meta-executive, utilities, and system interfaces necessary to run an experiment.

For each of these roles, resource al'-cation is one of the most important single problems in the design and implementation of LINGO. Resource allocation depends upon the ability to predict the amount of computing and channel usage that a particular experiment requires. This is important because the reliability and responsiveness of the system depend on such knowledge. Advance prediction of the system load will support rational scheduling and grouping of experiments.

It is important to realize that this assumes that the hardware and software will be in a reliable state so that rational resource allocation can be effective. An important element in that design is to decouple processing from real-time constraints as soon as possible within the ARC processing structure.

2.0 THE USER LANGUAGE DESIGN

This section contains a general description of LINGO with explanations and some of the alternatives considered for the critical design decisions. The formal specification of the LINGO language is in Appendix A.

LINGO provides a general, uniform way to describe a large scale computing process. Both real time and discrete processes can be described in terms of data movement and transformations that are to occur on the data. The LINGO language is designed primarily for disciplines in which a reasonable number of known functional transformations are applied in various combinations to process data. The language attempts to provide an easy safe way to interconnect these preconstructed functions. The following sections describe the rules by which data is transferred from the output of one transformation to an input of another. LINGO does not depend on a particular set of functions, but rather depends on an Application Library designer to supply a coordinated set of useful functions.

2.1 THE BENOIT MODEL

Previous work by Benoit yielded a first cut at how a user could picture an experiment. See Figure 2-1 for an example drawn from Benoit, et al [2]. Benoit modeled an experiment as a set of nodes connected by links. Each node represented a program such as an FFT. Each link represented a storage area into which the outputs of the previous node could be dumped and out of which the inputs to the next node could be retrieved in a first-come first-served fashion. The links connecting the nodes were divided into two classes: "data links" -- high bandwidth channels used to carry data, and "control links" -- low bandwidth channels used to pass parameters to nodes and to communicate status information. Data input to the system was modeled by a data source node. The experimenter was modeled by a control node that was the source of most of the experiment's parameters and the recepient of most of the experiment's results.

Using the Benoit model as a starting place this section will attempt to derive exactly what should be meant by "node" and "link," and what restrictions should be placed on an experiment in order to meet the design criteria developed in the previous sections. For convenience, these restrictions have been divided

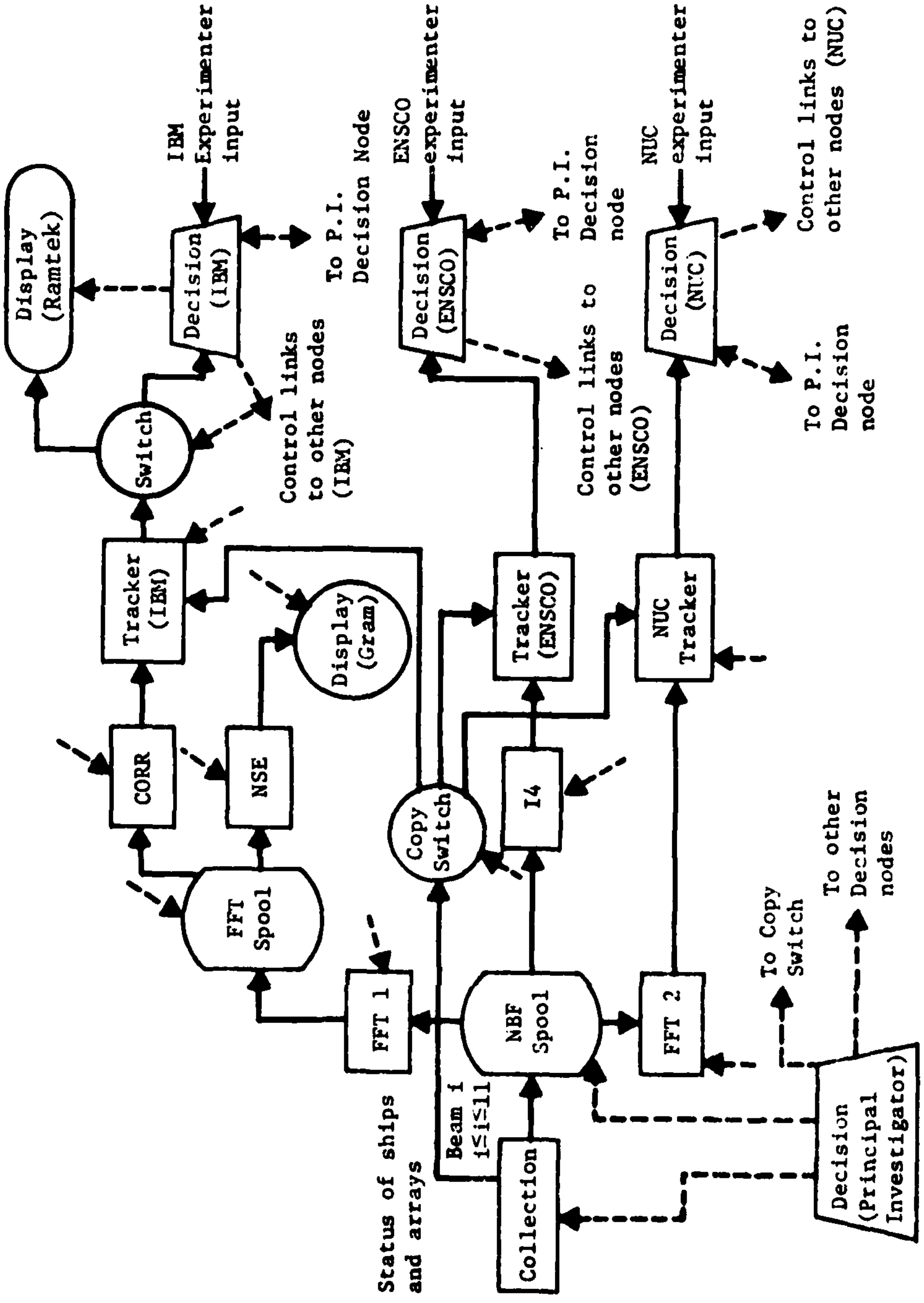


FIGURE 2-1 USER'S MODEL OF THE EXPERIMENT

into two classes: link-time restrictions that must be met before the experiment is run, and run-time restrictions that are checked while the experiment is in progress. For convenience a port is defined to be the point where a link enters (or exits) a node.

2.2 RELATIONSHIPS BETWEEN NODE INVOCATION AND INPUTS

Each node represents a program which presumably does some initialization and then performs a system call asking for input. When should the system return control to the node program in order to process the requested data?

The simplest form of an input request is a subroutine call that specifies which port to read and a location to store the resulting data. This possesses several undesirable traits. Deadlock could occur because two mutually dependent nodes may ask for data in the wrong order or they can be accidentally linked to some other node's outputs in the wrong manner. Since the subroutine call is executable and a part of the general flow of control within the node, it is unpredictable, in general, which parts will be used and in what order. In fact, the amount of data processed from one node relative to another cannot usually be determined, although for specific cases it could be predicted. In this environment, performance prediction and general behavior is hard if not impossible to verify at link-time.

Some restrictions are needed on the order and amount of input as well as how individual parts are handled. The following paragraphs describe the solution chosen for LINGO.

Consider a program node with N inputs. Of these, M are high bandwidth data and N-M are low bandwidth parameter ports. Because the high bandwidth ports actually get data that should drive the node, they are connected to other nodes by "running" links. The low bandwidth ports only give the node "advice," and are connected to other nodes by "memory" links. We can actually ignore the low bandwidth ports for purposes of determining when a node should be invoked.

Thus, two types of links are distinguished: The first is the high bandwidth or "running" link, and the second is the low-bandwidth or "memory" link. When a node is invoked it can use the data supplied by the memory port. If no data has arrived on the port's associated memory link since the last request for data, the memory port will supply the same data. The term "memory link" arises because an empty memory port always "remembers" the last data record it contained. Memory links must be initialized at the start of the experiment in order to prevent undefined results when a memory port is referenced for data. Thus, memory links are always guaranteed to supply some reasonable user defined data upon request.

If more than one block of data arrives at a port on either type of link, the data is queued for sequential processing by the node.

Most program nodes will require only one running link input. These nodes can be invoked when their running link contains data. When a program node requires more than one running input, there are more choices. For example, with two running inputs a node could be invoked when both inputs are available (called an ALL node), or it could be invoked when either link has input available (an ANY node). With more input links more choices are available. With M running links as input there are 2^M rules for invoking a node using an arbitrary AND/OR philosophy. Such complexity is undesirable and probably unnecessary because most nodes will not have more than one or two running input ports.

A first cut at simplifying the situation is to allow only pure ALL and ANY nodes. That is, an ALL node is invoked when all of its inputs are available and ANY node is invoked when one or more of its inputs become available.

ANY nodes can be replaced with a less complicated structure by considering how they would be used in an experimental situation. Most use of ANY nodes would be to merge data streams with no other processing; thus an experiment could replace such an ANY node with a simpler concept, the idea of merging links. When two links merge their data merges as well. Two such merging links must be carrying the same type of data so that the merged data stream is homogeneous. The advantage of merged data streams over ANY nodes is the ease in predicting their performance. The rate of flow in the merged stream is simply the sum of input flow rates. Merging links are used in LINGO to solve this problem, therefore, there is only one type of LINGO node: those that correspond to ALL nodes in this discussion

A network with only ALL nodes and merging links can still deadlock and such deadlocks can be prevented with the restriction that loops of running links are not allowed. If a loop of links is desired, at least one of them must be a memory link. This link-time restriction makes deadlock impossible because the memory link in a loop effectively cuts off the synchronization which makes the deadlock possible. These restrictions are employed in LINGO to prevent deadlock.

2.3 RELATIONSHIPS BETWEEN NODE EXECUTION AND OUTPUTS

Certain experiment networks are undesirable because they imply infinitely growing queues on data links. The simplest example is a node that is accepting real-time input at a rate faster than it can process. Such a situation can be avoided by careful attention paid to performance prediction at link-time. For example, any real-time inputs to an experiment will have data rates known by the LINGO linker; similarly the standard LINGO library nodes will have been benchmarked so that the linker can predict whether any experiment constructed entirely from library nodes will be feasible. User-constructed nodes are handled by allowing the user/experimenter to specify a maximum processing time for a node which will be enforced by the system at run-time. If the input to the experiment is arriving from an archive or a spool, the input rate can be adjusted at run-time to fit the processing rate of the experiment.

The other major problem with infinite queues is caused by a node where one of its input links has data arriving at a faster rate than some of the others. The data on the fastest link will queue infinitely. This situation is prevented by scheduling nodes so that the input rates of all links into a node will be equal so that no queues can build up on a link.

This need to predict the behavior of links implies a need to know the amount of output a node generates for each invocation on each of its output ports.

Detecting data rate errors requires that the compiler knows exactly how much output a node will produce for each invocation on each of its output ports. For library nodes, this will be done by understanding them and describing their behavior; for user/experimenter nodes, this can be done by allowing the user/experimenter to specify the outputs per invocation and then checking the veracity of the specification at run-time. Such precise knowledge of the output/invocation ratio is needed only when many links are destined for an input to a common node; with most experiment networks this restriction will not even be noticed.

2.4 CONTINUOUS VS. BLOCKED DATA

Experimental data may be split into two classes: continuous data, such as time data input to an FFT, in which the data consists of small logical pieces blocked physically for the operating system's convenience; and blocked data, such as the output from an FFT, in which the data is already logically blocked. Note that when two data links are merged, any single invocation from the merged link will obtain data only from one of the data links; thus blocks are not broken up and continuous streams are not intermingled within a given invocation.

Continuous data causes problems in analysis because a program node may need varying amounts of it depending on other parameters. For example, an FFT may have a memory link input telling it how much time data to "bite off" at each activation. In such cases it is helpful to specify at link-time a minimum and maximum of the amount to be bitten off; in most cases the minimum should equal the maximum.

Performance for continuous data depends on the size of the input block, the amount the input block should overlap the previous block, and the sampling rate. For performance analysis purposes, these three statistics should be specified at link-time, but if run-time parameter changing is desired the solution described above may be employed.

2.5 END OF FILE AND PARAMETER MODIFICATION

Eventually all experiments must stop. This can happen because the system crashes, because the user/experimenter wishes to switch to a completely different LINGO experiment, or because the same experient is to be run again with a different organization. This document will ignore the first cause which is treated in the METAXEC description [8] and concentrate on "normal" transitions in LINGO experiment definitions.

Suppose a LINGO experiment is about to come to an end. This can be modeled by placing an end-of-file mark on each input running link at some appropriate time. When an EOF mark reaches a (single running input) node, it will cause the node to do cleanup processing and, after its completion an EOF mark will be placed on all its (running) output links. As EOF marks arrive at ports they deactivate the port. Eventually EOF marks will reach all the running input links to a node and end-of-file processing will be initiated. Note that when two links merge that an EOF mark is found on the merged link only after both links have reached an EOF.

The above method can be pictured as using the EOF marks to flush the old LINGO experiment out of the system. If the new LINGO experiment resembles the old one topologically with only a few parameter changes, these parameters could fall in a special record which follows the EOF marks of the old experiment and which leads the data of the new experiment. This method would allow parameter changes to be synchronized cleanly from node to node such that if many parameters are to be changed, the changes all fall on the next piece of new data rather than haphazardly on all the data currently in the system. Asynchronous parameter changes can be accomplished directly via memory links from the user/experimenter to each node.

3.0 USER/EQUIPMENT INTERFACE

This section contains a description of the user/equipment interface required to support the LINGO language. The phased development planned for the language significantly affects the need for various types of equipment to support the user interface. Definition of the phases of development and allocation of functional capabilities to the phases has not yet been determined in enough detail for complete specification of the user interface equipment, especially the terminals that will be required to support the graphical form of LINGO. However, it is clear that standard, typewriter-like terminals, e.g., Teletype, can support the linear character form of LINGO. As more detailed design is completed and the development phases are more sharply defined, the requirements for user interface equipment will be more completely specified.

3.1 CHARACTER FORM OF LINGO NETWORKS

The syntactic portions of Appendix A prescribe the way in which a LINGO network is to be expressed in character form. Figure 3-1 shows an example LINGO program using this character representation. The example is only included here to demonstrate the character form of the user/equipment interface; the content of the example is explained in Section 9.2 of Appendix A.

The character form of LINGO has been carefully designed to stay within the ASCII 128 character set (see Figure 3-2). However, it is possible to specify LINGO networks in the ASCII 64 character set which excludes the lower-case characters and some special symbols.

3.2 GRAPHICAL FORM OF LINGO NETWORKS

The most natural way for a user to employ LINGO is to draw nodes and links so that the network becomes a visual entity. In the future, a graphical LINGO linker will be defined so that users may have mechanical interactive help in constructing networks. Appendix A specifies a standard graphical representation for the user portion of LINGO networks. The following sections summarize that material and provide a general example. Figure 3-3 shows the same example program used earlier for character representations in the simpler graphical form. This demonstrates how a graphical, interactive LINGO linker would represent LINGO networks.

FIGURE 3-1 AN EXAMPLE LINGO NETWORK USING
THE STANDARD CHARACTER REPRESENTATION

```
def lib matpak; {include matrix library}

node grad_t_f =
    import vector (150) f;
    export vector (150) del_t;
    begin env ("/u/dt/gradtf",
               fortran,
               pdp10,
               1000 ms,
               del_t:  1 out/inv);
    end;


import matrix (150, 150) pq,
       vector (150) f;
export file g;


begin
    copy port f --> f1, f2;
    times (port pq, grad_t_f(f2))--> a;
    times (<scalar(/s = "0.5"), port h>, a)-->b;
    display (minus(f1,b)/format="...")--> port g
end
```

FIGURE 3-2 USA STANDARD CODE FOR INFORMATION INTERCHANGE

USA Standard Code for Information Interchange

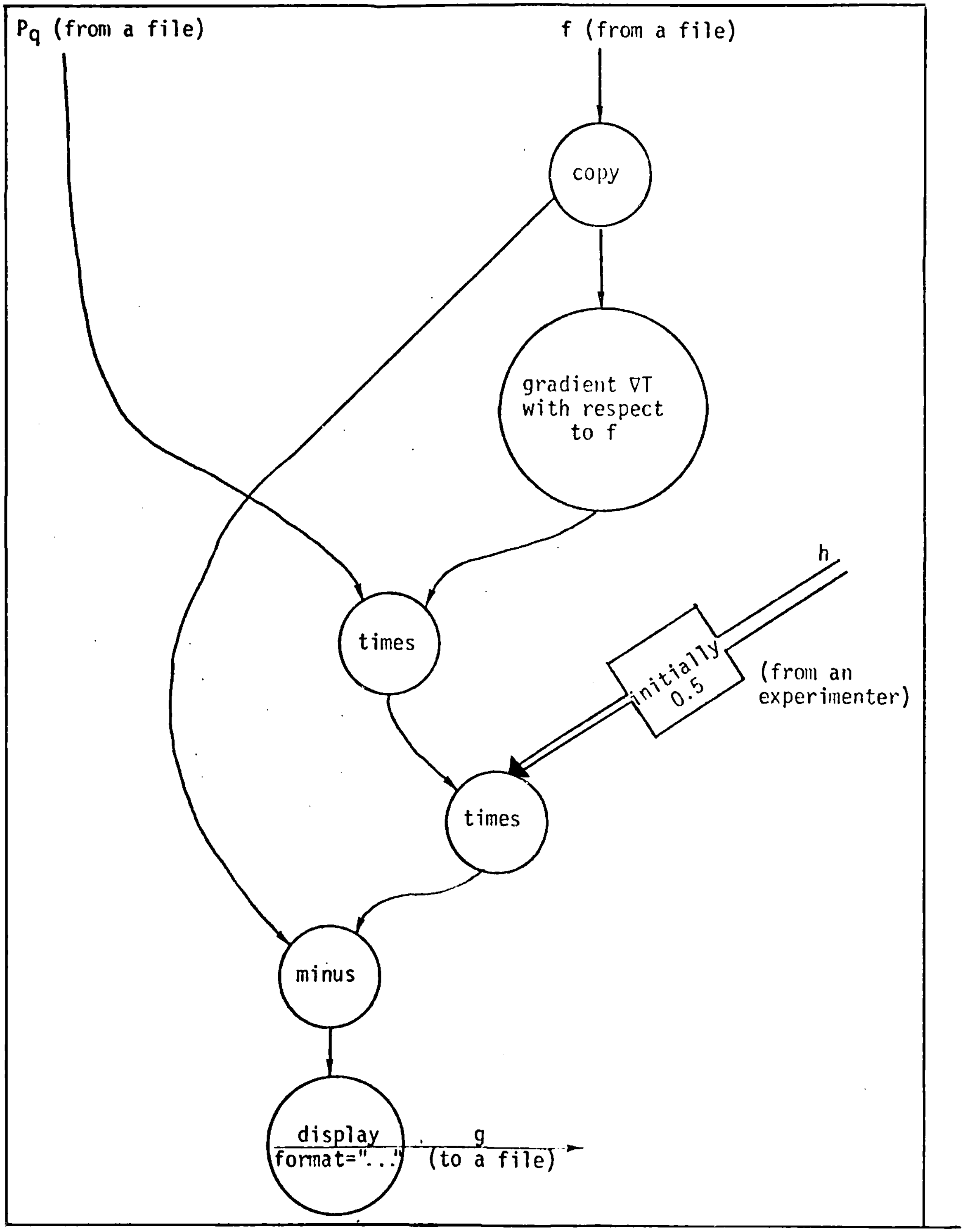
1. Scope

This coded character set is to be used for the general interchange of information among information processing systems, communication systems, and associated equipment.

2. Standard Code

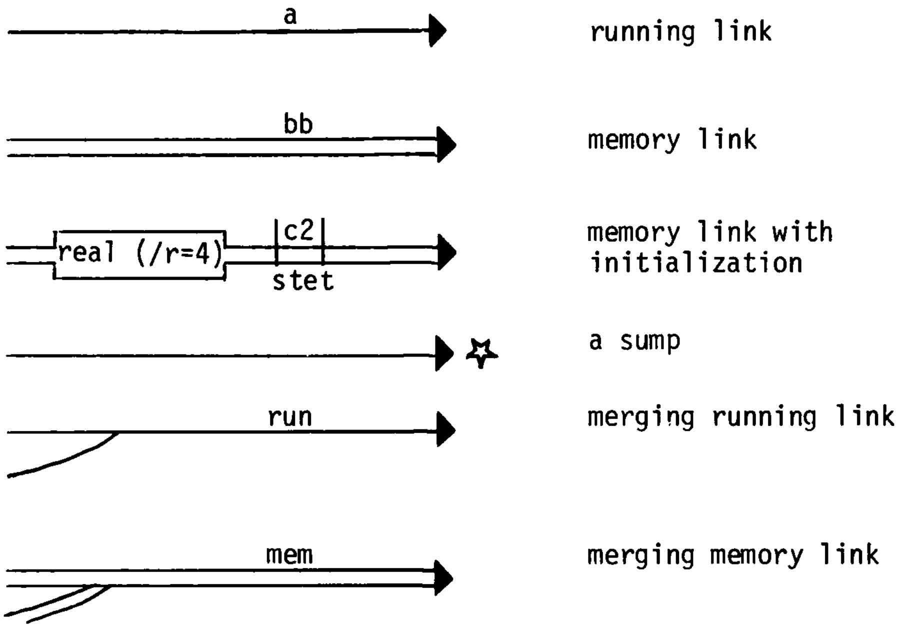
Bits b7 b6 b5					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	COLUMN / ROW	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	−	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

FIGURE 3-3 NETWORK FOR $g = f - HP_q \nabla T$



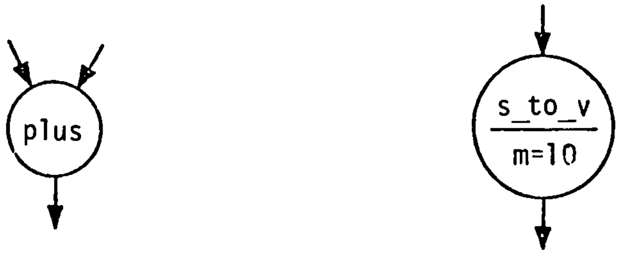
3.2.1 Graphical Form of Links

A running link is represented by a single line and a memory link by two parallel lines. Merging links are represented by merging lines and a sump by an asterisk. If a link has a link-id, the identifier is written on or next to the link. Thus:



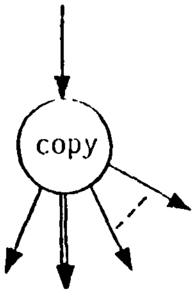
3.2.2 Graphical Form of Nodes

A node is represented by a circle with imports as arrow inputs to the circle and exports as lines out. The name of the node is written in the circle. If modifiers are present they are written in the circle separated from the node name by a line. Thus:



3.2.3 Graphical Form of Copies

A copy is represented by a circle with one input and any number of outputs. The circle is named "copy".



4.0 COMPUTER PROGRAMS

This section contains a specification of the computer programs required to support LINGO. At the current stage of design, only general specifications of the requirements for these programs can be determined; more complete specification of the requirements will be completed when the prerequisite details of the METAEXEC and hardware configuration have been determined.

4.1 LINGO LINKER

A computer program will be required to process LINGO to produce linking directions for the METAEXEC. This program will have three inputs:

1. Control commands specifying parameters which control its mode of operation.
2. An experiment network specification in the experiment design language.
3. Libraries of node and subordinate network specifications in the experiment design language.

The LINGO linker will have two outputs:

1. A listing or display of the results of analyzing the experiment network specification. These results will consist of error reports and performance predictions covering space and processor time and channel utilization.
2. Control tables suitable for use by the METAEXEC during operation of the experiment.

The functional capabilities required of the LINGO linker are:

1. Retrieval of referenced node and network specifications from library files.
2. Syntactic analysis. Syntactic errors will be reported with meaningful messages.
3. Semantic analysis. Semantic errors will be reported with meaningful messages derived from additional data supplied by the node specifications from the library.

4. Analysis of resources required for operation of the experiment and estimates of the performance of the experiment. The results of this analysis will be reported to the user in a form that both supports evaluation of the load the experiment will place on the system and support estimation of the feasibility of operating the experiment at the same time as other experiments or with less than the complete hardware configuration.
5. Production of tables to control operation of the experiment. These tables must be compatible with the detailed design of the METAEXEC system software and the interpreter of the experiment control language.

Operation of the LINGO linker will not be subject to real time constraints so there will not be hard constraints on its speed of operation.

4.2 EXPERIMENT CONTROL LANGUAGE INTERPRETER

A program will be required to interpret the experiment control language. The requirements of this program cannot be properly determined until completion of the language design, and a more detailed level of design of the METAEXEC system software is available.

1. The interpreter will be an interactive program.
2. The user must be able to refer to elements of the experiment network in terms that are similar to those used in specifying that network.
3. The speed of operation of the interpreter must be compatible with the real time requirements of experiments.
4. The interpreter must be capable of supporting testing and debugging operations during experiment development.

4.3 AUXILIARY LANGUAGE PROCESSORS

It is expected that there will be a need for additional utility processing programs to help support LINGO. One candidate function is management of the libraries of primitive node programs, node/network specifications and experiment control tables. It is not yet clear whether there should be a distinct program for this function or whether it should be distributed among the programs specified in Sections 4.1 and 4.2. A second candidate function is the process of loading and initializing an experiment. As the system design is carried to a greater level of detail, specific requirements for these functions will be determined and they will be allocated to system components.

5.0 TEST AND VERIFICATION

Computer languages and language ideas are generally tested for logical consistency, usefulness and user acceptance. In the design of LINGO several test and verification steps were undertaken and several remain to be done.

Logical inconsistencies of language features can be revealed by a formal specification of the syntax and semantics of the language. Formal syntactic specification will reveal errors in the general organization of the language and the proper coupling of semantic concepts; however, it will not test for ambiguity and deeper structural errors such as scope rule inconsistencies. LINGO has been formally specified syntactically and the logical organization of the language has been verified in terms of the related semantics. The LINGO syntax has not been processed by a parser generator which would provide yet another test of its coherence and logical consistency.

LINGO semantics have not been formally specified; rather, English was used as the semantic specifier. Testing of the semantics could be done by implementing the LINGO linker which would force the degree of understanding of the semantics necessary to clean up any remaining bugs.

The utility of language features is a more intuitive concept than that of logical consistency. To provide validation of this concept an attempt to judge the generality-utility cross product was done by critical evaluation of the LINGO concepts by the implementation team. Independent judgment of "critical users" was solicited by a presentation of the language concepts for review. In every case these efforts led to an improved language. In the last analysis the judgment of the user community will be the final verification of the usefulness of the features included. Evaluation of this can be assisted by using an automated complaint log for user comments.

User acceptance of a language is almost impossible to verify in the absence of a language processor, so until an implementation of a LINGO system is accomplished this goal is impossible to verify.

6.0 REFERENCES

[1] American National Standards Institute, Inc. USA Standard Code for Information Interchange. USAS X3.4-1968.

[2] Benoit, J., N. Chowla, R. Mikelskas, W. Pailen, and A. Williams. ARC User Interface and METAEXEC Design Considerations. The MITRE Corp., (MITRE Technical Report MTR-7511), April, 1977.

[3] Bourne, S., A. Birrell, and I. Walker. ALGOL68C Reference Manual. Cambridge, England: Cambridge University, 1972.

[4] Cleaveland, J. C. Pouches, a programming language construct encouraging redundancy. Computer Science Dept., UCLA, (UCLA-ENG-7555), 1975.

[5] Dennis, J. B. First Version of a Data-flow Procedure Language. MIT Project MAC, Computing Structures Group Memo 93-1, Aug. 1974.

[6] Gardner, R. I. A Methodology for Digital System Design based on Structural and Function Modeling. Computer Science Dept., UCLA, (UCLA-ENG-7488), 1975.

[7] Hamlet, R. G. High-level Binding with Low-level Linkers. Communications of the ACM, 19, 11, 642-644 (1976).

[8] Hinke, T. H., and C. M. Switzky. ARC System Software Specification. SDC, TM-5898/000/00, June 1977.

[9] Kleinrock, L. Analytic and Simulation Methods in Computer Network Design. Proceedings of the Spring Joint Computer Conference, Atlantic City, N.J., 569-579 (1969).

[10] Lindsey, C. H., and S. van der Meulen. Informal Introduction to ALGOL 68. rev. ed. Amsterdam: North-Holland Publ. Co., 1977.

[11] Martin, D. F., and G. Estrin. Experiments on Models of Computations and Systems. IEEE Transactions on Computers, EC-16: 59-69 (1967).

[12] Meertens, L. Mode in Meaning. New Directions in Algorithmic Languages. Institut De Recherche D'Informatique et D'Automatique, France, 1975.

APPENDIX A

PRELIMINARY REPORT ON THE

NETWORK LANGUAGE LINGO

PRINCIPAL AUTHORS

PAUL R. EGGERT
ROBERT C. UZGALIS

TABLE OF CONTENTS

PAGE

PAGE

A tale should be judicious, clear, succinct;
The language plain, and incidents well link'd.

--Conversation

Thousands ...
Kiss the book's outside who ne'er look within.

--Expostulation

William Cowper (1731-1800)

1.0 STRUCTURE CF THE LANGUAGE

1.1 GOALS

The Network Language, LINGO, was designed to:

- Provide a notation which completely and concisely defines a system composed of several programs operating in concert on one or more machines.
- Provide for the creation, maintenance, and use of libiaries of programs where each library represents a coherent set of data types and parameterized nodes which operate on those types.
- Allow designers of node libraries to detect and announce misuse of their designs as their nodes are linked into the user-defined network rather than issue an error message after the network begins execution.
- Prevent deadlocks in user networks. A LINGO network which executes on an idealized infinite-queue machine will never deadlock; any implementation with limited buffers can detect potential trouble spots at link-time before a network begins execution.
- Allow precise estimates of requirements of system resources, particularly buffer space and processor time, to be provided by the network linker prior to execution of a LINGO network.

1.2 NODES

LINGO is the control language for a general purpose linker for independent programs called nodes. A node is defined by writing a program in some algorithmic language such as Fortran, Algol, PL/I, or Assembler, or by writing a network in LINGO which connects other nodes to perform the desired function.

A node can be conceived of as an abstract function that operates on a set of data records that are present at the node's inout ports. The node executes and produces a series of data records on its output ports. Eventually the node will finish and go to sleep awaiting more data on its input ports. For brevity in the rest of this document input ports will be called "imports" and output ports, "exports".

Routines designed to be interconnected in a LINGO network and written in an algorithmic language are called nodes. Primitive nodes are described within LINGO by environment information which includes performance data, the name of the executable program module, the location of the program, and on what machine the program will execute. Performance data includes the amount of storage the program requires, the number of buffers required, and estimates for the amount of time each routine needs for every invocation.

1.3 LINKS

Links are used to connect an export of one node to an input of another. Records enter a link at one end and exit at the other end in the same order as they were put on the link; thus a link is a first-in, first-out queue. Within a LINGO network there may be many independent link entries that feed records to a common destination. Records are never lost or overwritten on a link. Since the processing at the destination end may be slower than the processing at the source end of the link, data records may accumulate on the link during processing. However, any link may be implemented by using a finite amount of storage known in advance of executing a network.

LINGO was designed to connect together nodes operating on several independently operating machines. In specifying a network the user links nodes together to perform a task. A link which the user specifies to connect nodes is a path upon which data flows from the export of one node to an import of another. Since nodes may be operating on independent machines, the links must provide the capability of synchronizing the node actions within the network. Because of variance in the node processing capabilities and how nodes are physically linked, a LINGO user may not assume how much time is taken by a link to transmit data records or how soon records will be processed at the receiving end.

To help a user synchronize node activity, LINGO provides two types of links: running links and memory links. The only difference between them is the action they take when no data has arrived at the destination node. A memory link always remembers the last data record. This remembered record is supplied when a node requests data when no new data has arrived on the link; thus, the memory link acts asynchronously with respect to other links. Running links are always synchronized and a node waits for data records to be present on all of its running input links before it will begin execution. Thus, data on memory links does not participate in deciding when a node is to run.

Memory links are useful for transmitting slowly varying control information or parameters to a node from either another node or another source, such as an experimenter's terminal. Running links are useful for transmitting data to be operated on by a network. Thus, a LINGO network is a data-driven system, where the driving data is on running links and asynchronous control is effected by memory links.

To prevent deadlock from a circular wait condition, running links may not form a loop. Thus, if only running links are considered, a LINGO network is purely a feedforward system. Memory links may loop back because they are not used for synchronization and so cannot cause deadlock. If a portion of an algorithm requires synchronous feedback, it may be written in an algorithmic language and included as a primitive node.

1.4 NETWORK DATA

The data records to be transported on network links can assume an arbitrary form that is fixed when the network is linked together. Each node specifies a general configuration for data acceptable at each port. When a node is linked into a network, the network linker checks to see that the specific form of data flowing on each input link is compatible with the general configuration of the import into which it flows. Once a node is linked into a network, the network linker can determine the specific form of data flowing out of each export by examining the import types, modifiers supplied by the user, and the LINGO specification of the node.

1.5 NODE LIBRARY

LINGO depends heavily for its utility on the designer of the nodes that it will eventually link. The node designer must conceive of reasonable functions and data types that are well suited to solving problems in some application area. For example, a library of application nodes could be defined using some standard form of collected data to perform signal processing applications, statistical analysis, or report generation.

To help the node designer assist his users, LINGO allows him to specify a set of computations to be done when a node is being linked into a network. This allows the node designer to check properties that are being employed in the node against specific properties of the input type on an import. Node-dependent diagnostic warning and error messages can then be issued by the node library via the network linker to the user.

1.6 REPRESENTATION ISSUES

The most natural representation for a LINGO network is a graph, with circles representing nodes and arcs representing the links between them. This representation makes the processing very clear and it is a nice conceptual aid. However, a linear character representation is also useful to allow input from any kind of terminal. LINGO has been designed with both of these forms in mind and both are specified in the remainder of the document.

1.7 NOTATION FOR SYNTACTIC DESCRIPTION

Grammatical notations in this document are expressed in a somewhat simplified form of Backus-Naur Form (BNF). This simplified form represents nonterminal grammatical notions using a sequence of lower case letters. Thus, the BNF notion "<program body>" would be written "program body" in the simplified notation. Terminal grammatical notions in BNF are written as is, but here they will be represented by non-terminal notions which end in "symbol". Thus, in BNF a comma between two items would be represented by "," in the grammar whereas in the simplified notation it is represented by the notion "comma symbol". Section 7 contains a list of all the LINGO symbols and their corresponding typographical representation.

The remaining conventions used in specifying the syntax of LINGO is easy to define by substituting a corresponding BNF rule. The BNF production symbol "::=" is replaced by ":" in the simplified notation. The BNF or-bar "|", separating alternatives, is replaced by a semicolon ";" in the simplified notation. In BNF, two grammatical notions written in sequence indicate concatenation of the terminal productions from the notions. This is represented by separating the notions by a comma "," in the simplified notation. The end of a grammar rule is indicated by a period "." in the simplified notation.

To read a simplified grammar rule, substitute the following phrases for the indicated punctuation:

":"	is a(n)
","	followed by a(n)
";"	or it is a(n)

Thus the rule:

```
program:  program identifier, equals symbol, program body;
          program body.
```

is read "A program <u>is a</u> program identifier <u>followed by an</u> equals symbol <u>followed by a</u> program body <u>or it is a</u> program body." This same rule would have been written in BNF:

```
<program> ::= <program identifier> = <program body> |
                 <program body>
```

To ease the number of rules that need to be read to understand the intent of the grammar several abbreviations are used. These abbreviations take the form of several standard suffixes to a grammar notion. The suffixes include "list", "option", "pack", etc., which have the meaning indicated in Table A-1. For example, the notion "argument list" indicates that the grammar notion "argument" will be repeated an arbitrary number of times separated by comma symbols. The notion "argument option" indicates the "argument" is optional and can be omitted. Composites are possible; thus "argument list option" indicates that the entire list of arguments may be omitted including the commas separating members cf the list, whereas "argument option list" indicates that some of the arguments may be omitted but the commas forming the list will still be present even though a particular argument is gone.

Using these conventional suffixes the earlier example rule for a program could be rewritten:

```
program:  program idf option, program body.
program idf:  program identifier, equals symbol.
```

To properly define where comments, blanks, and other typographical display characters (like newline, tab, etc.) can be placed, another conventional suffix is used. Notions which end in "token" can be preceded by comments, blanks, etc. Notions which end in "symbol" represent only the single associated typographical character. These suffix rules are formally defined in Section 7 of this appendix.

1.8 ORGANIZATION OF THE LANGUAGE DEFINITION

This appendix treats the larger syntactic forms first, and then progresses to smaller and smaller entities. Thus, a network-program is presented first, while the section which defines the characters and symbols used in writing LINGO networks is last. Sections 2 through 7 treat network and definition statements, links, data paths, network data, computations at link time, and finally the symbols and identifiers used to define the rest. Section 8 is an

TABLE A-1 SUFFIX FORMS USED IN THE GRAMMAR

SUFFIX	FORM
list	item, item, ..., item
option	item or EMPTY
pack	(item)
group	item; item; ...; item
sequence	item item item ... item
id	an identifier
token	a notion signifying some typographical glyph which may be preceded by comments, blanks and other typographical formatting characters
symbol	a notion which signifies some typographical glyph defined in Section 7

informal discussion that defines what a well behaved node is permitted to do and how the node can interact with the LINGO run-time facilities. Section 9 presents a small sample-library of primitive nodes and a LINGO program using them.

The top-down organization of this document is an advantage because it generally presents the important information first. However it has one major drawback: Since larger entities are always defined in terms of smaller ones, examples of larger entities use constructs which will be formally defined later in the document. Sometimes the example involves syntax so far removed that it was thought better not to confuse the reader with an example. In these cases, ellipses (...) are placed in the portion of the example which would offend.

The semantics for LINGO are defined in English and are thus relatively informal. Within the descriptions of semantics a grammatical notion is written with hyphens between words; thus if "actual record type" appears in a grammar rule, it is referred to in the semantics as "acutal-record-type". Grammar notions are written in the plural by adding an apostrophe followed by an "s". Thus "actual-record-type's" is the plural.

Care has been taken to be as precise and complete as possible without relying upon a particular implementation. For this reason it will be noted that the internal notions of LINGO are specified more precisely than the parts of LINGO which interface to the system which implements it. Once that system is defined, an implementor can make the external notions of LINGO more precise.

2.0 NETWORKS AND DEFINITIONS

2.1 NETWORK PROGRAMS

2.1.1 Syntax

a) network program: network idf option, external node network.

b) network idf: network id, is token.

c) external node network: external import part option, external export part, network body.

d) external import part: def prelude option, import token, import info list, with token.

e) import info: actual type, import id, file env option.

f) external export part: def prelude option, export token, export info list, with token.

g) export info: actual type, export id, file env option.

h) file env: environment token, file name list pack.

i) file name: string expression.

j) def prelude: definition token, definition group, with token.

Examples

```
a)  fme_c =   def lib   arcfme;
              import    mpx remote,
                        line director,
                        line sysin env ("/dev/terminal");
              export    line sysprint;
              begin     ... end
```

```
b)  fme_c =
c)  def lib   arcfme;
    import    mpx remote,
              line director,
              line sysin env ("/dev/terminal");
    export    line sysprint;
    begin     ... end
d)  def lib   arcfme;
    import    mpx remote,
              line director,
              line sysin env ("/dev/terminal");
e)  line sysin env ("/dev/terminal")
f)  export    line sysprint;
g)  line sysprint
h)  env ("/dev/terminal")
i)  "/dev/terminal"
j)  def lib arcfme;
```

2.1.2 Semantics

A network-program consists of five things:

1. An optional network-idf that specifies its external name.
2. Optional def-prelude's that gives preliminary definitions either directly or from a library.
3. An optional list of import-info's, each of which specifies the connection between a network-program import and an external resource. Such a resource must have a type that matches the import-info's actual type. The optional file-env can be used by an implementation for additional information. For example, it could be used to locate a specific file or a set of concatenated resources. If the file-env is absent then a file-name derived from the import-id is assumed.
4. A list of export-info's, each of which specifies the connection between a network-program export and an external resource. An export-info is like an import-info except that it applies to output rather than input.
5. A network-body, which will be defined in Section 2.2.

External resources are connected to an import by successively transporting logical records from the resource to the associated port. This continues until an end-of-file is sensed at an import. At this time the port becomes inactive and an end-of-file is placed on the associated link.

The action of a network-program is the activation of its external-node-network. The network-program is active when its external-node-network is active and conversely. When the external-node-network is no longer active, an end-of-file is transmitted to the external resources connected to each export, and the network-program terminates by returning control to the operating system.

The scope of import- and export-id is the network-body portion of the external-node-network in which the import- and export-id are defined.

2.2 NETWORKS

2.2.1 LINGO Node Networks

2.2.1.1 Syntax

a) network body: begin token, data path group,
with token option, end token;
begin token, primitive node spec,
end token.

b) definition: node def; record type def; value def;
library inclusion; use check.

Examples

a) begin ... end

a) begin env ("gradient", assembler, pdp10, 500 ms, v:1 out/inv) end

2.2.1.2 Semantics

The network-body is the active part of a network-program. Each data-path in its data-path-group acts independently and simultaneously. Therefore, data-path's can be given in any order. When activated, a network-body activates all data-path's in the data-path-group. A network-body is active whenever any one of its data-path's are active. A network-body terminates when all of its data-path's are inactive. If a network-body contains a primitive-node-spec, it activates the program associated with that primitive-node-spec. It terminates when the primitive-node program has finished processing an end-of-file invocation. Activation and termination are LINGO technical terms which indicate the capacity or incapacity to process records.

Definitions specify the identifiers associated with nodes, record types, and values. They also specify the inclusion of libraries containing additional definitions to be used, as well as use-check's to warn about improper linkage. The scope of an identifier defined in any definition is that portion of the containing (external-)node-network which follows the definition. The scope rule thus prevents referencing an identifier before it has been defined.

2.2.2 Primitive Nodes

2.2.2.1 Syntax

a) primitive node spec: environment token, primitive info pack.

b) primitive info: object file name, comma token,
language name, comma token,
machine name, comma token,
ms per invocation, comma token,
output spec list.

c) object file name: string expression.

d) language name: string expression.

e) machine name: string expression.

f) ms per invocation integer expression, ms token.

g) output spec: export id list, colon token,
output characterization option,
output ratio.

h) output characterization: worst token.

i) output ratio: outputs per invocation;
invocations per output.

j) outputs per invocation: integer expression, output token,
slash token, invocation token.

k) invocations per output: integer expression, invocation token,
slash token, output token.

Examples

```
a) env ("/dt/mat/gradient",
        fortran,                {it is assumed that "fortran" and "pdp10"}
        pdp10,                  {are previously defined value ids        }
        30 + 150*m ms,
        v2:  1 out/inv)
b) "/dt/mat/gradient",
   fortran,
   pdp10,
   30 + 150*m ms,
   v2:  1 out/inv
c) "/dt/mat/gradient"
d) fortran
e) pdp10
f) 35 + 150*m ms
g) v2:  1 out/inv
h) worst
i) 1 out/inv
j) 1 out/inv
k) n inv/out
```

2.2.2.2 Semantics

A primitive-node-spec gives information to the network linker about a primitive node written in an algorithmic language. Its object-file-name gives the location of the object module that implements the primitive node. The language-name identifies the source language and thus the particular linkage conventions of the object module. The machine-name identifies the machine on which the object-module will execute. The ms-per-invocation provides an estimate of cpu time, exclusive of system overhead, that the node consumes on every invocation. The output-spec provides the number of outputs expected from each export for each invocation.

Every export-id (including external-export-id's) of the export-part of the network-body (or external-node-network) containing the primitive-node-spec must appear once and only once in the output-spec. Each output-spec contains three things: a list of export-id's, an optional output-characterization giving output-spec accuracy information, and an output-ratio that specifies the amount of collective output that may occur on the exports identified by the export-id-list.

An output-characterization, if present, implies that the total amount of output per invocation is unpredictable but will never exceed the amount specified in the output-ratio. If the output-characterization is absent, the total amount of output per invocation will always be the amount specified in the output-ratio.

An outputs-per-invocation specifies that the output-ratio is in terms of the number of outputs that are yielded by or arbitrary invocation of the node. Conversely, an invocations-per-output specifies that the output-ratio is in terms of the number of invocations expected to produce a single output.

2.3 NODE DEFINITIONS

2.3.1 Node Definition

2.3.1.1 Syntax

a) node def: node token, node id, is token,
node network.

b) node network: import part, modifier part option,
export part, net work body,

Examples

```
a)  node grad =   def lib  matlib;
                  import   vector (mod n) f;
                  export   vector (n) del_f;
                  def      unless 0 < n & n < = 4096
                              error ("grad:" # n #" out of range");
                  begin env ("/dt/mat/gradient", fortran, pdp10,
                           50 + 100*n ms, n:1 out/inv) end
b)  import vector (mod n) f;
    export vector (n) del_del_f;
    begin grad (grad (port f)) --> port del_del_f end
```

2.3.1.2 Semantics

A node-def defines the properties of a node and identifies the node with its node-id. When a node-id is referenced in a node-invocation, it is identified by its name and input types as described in Section 4.2.2. After it is identified, its modifiers are determined from the node-modifiers of the node-invocation, the defaults from its modifier-part, and the types of its exports from its export-part. In doing this the network linker evaluates all of the definitions within the network-body and issues any error messages given in any use-checks. The node is then considered linked into the network.

The scope of an identifier defined outside the node-network, which is the same as an identifier defined within the node-network, does not extend into the node-network.

2.3.2 Node Imports and Exports

2.3.2.1 Syntax

a)	import part:	def prelude option, import token, import spec list, with token.
b)	import spec:	formal type, export id.
c)	export part:	def prelude option, export token, export spec list, with token.
d)	export spec:	actual type, import id.

Examples

a) import line a, line b

b) line a

c) dim (1:10) line c

d) export line d, line e, line f

e) struct (real, line, dim (1:10) integer) diagnostic

2.3.2.2 Semantics

The import-part and the export-part together specify the number of imports and exports and the type associated with each port. The formal-type in an import-spec identifies a class of record-types acceptable to the corresponding import and defines modifiers that have the value from the matching position in the record-type connected to the port.

The actual-type in an export-spec must employ identifiers that will be known at the time the node is linked. The actual-type specifies computations that allow the network linker to compute the complete record-type for export links.

2.3.3 Modifiers

2.3.3.1 Syntax

a) modifier part: modifier token, modifier spec list.

b) modifier spec: modifier id, default option.

c) default: is token, static expression.

Examples

a) mod a, b, c = 13, d = true, fmod

b) aname

b) aname = 13*a

c) = 13* a + b

c) = 27

2.3.3.2 Semantics

A modifier-part is performed by defining each modifier-id in the modifier-spec-list with a preliminary value. A modifier-spec is performed by the network linker by evaluating the static-expression of its default and assigning the resulting value to its modifier-id. If the default-option is empty, the modifier-id's value is undefined.

The scope of the modifier-id defined in a modifier-spec extends to all succeeding parts of the node-network that contain the modifier-spec.

2.4 RECORD TYPE DEFINITIONS

2.4.1 Syntax

a) record type def: record token, record type id,
modifer id list pack option,
is token, string expression.

Examples

a) record vector (n) = "dim (1:" # n #") real"

a) record line (n) = "struct (integer, dim (1:" # n #") char)"

a) record terminal = "line (80)"

2.4.2 Semantics

A record-type-def defines a record-type-id for refering to a parameterized record type. This allows a shorthand for data descriptions either of formal-record-types or actual-record-types.

The string-expression is in terms of modifiers whose scope extends to the position and local modifiers defined in modifier-id-list-pack where the string-expression will be expanded. The scope of these local modifiers is the string-expression. The resulting string value must conform to a formal-record-type or an actual-record-type after expansion, depending on the context.

2.5 VALUE DEFINITIONS

2.5.1 Syntax

a) value def: value id, is token, static expression.

Examples

a) k = a*10

b) message = n # "too large"

2.5.2 Semantics

A value-def is evaluated by the network linker by evaluating its static-expression and associating the resulting value with the value-id.

2.6 LIBRARY INCLUSIONS

2.6.1 Syntax

a) library inclusion: library token, library spec list.

b) library spec: library id,
library modifications pack option.

c) library modifications: library modifier list.

d) library modifier: modifier id, is token, static expression.

Examples

a) lib arclib, fme_ib, stab_lib(ncases = 500)
b) arclib
b) stat_lib(ncases = 500)
c) ncases = 500, missing = true
d) ncases = 500

2.6.2 Semantics

A library-inclusion specifies a series of external library names that are to be included in this node-network in the order specified.

A library is a modifier-part-option followed by a def-prelude. Note that a library may only contain definition's; no data-path's, import or export specification, nor primitive-node-spec's may appear in a library unless it occurs within one of its definition's.

A library-inclusion is evaluated by the network linker by replacing it with a modified textual copy of the def-prelude portion of each library specified in the library-inclusion library-spec-list. If there is more than one library, the modified def-prelude's should be separated by with-token's. The result of replacement is a group of definition's and use-check's each of which is evaluated by the network linker.

The textual copy of a library's def-prelude is modified by replacing each occurrence of each modifier-id in the def-prelude with the value that resulted from evaluating the static-expression corresponding to that modifier-id in the library-modifications of the library-spec that specified that library.

2.7 USE CHECKS

2.7.1 Syntax

a) use check: unless token, boolean expression,
diagnostic.
b) diagnostic: diagnostic severity,
string expression pack.
c) diagnostic severity: warn token; error token; abort token.

Examples

a) unless k > 100 warn ("storage problem")

b) warn ("storage problem" # k)

c) warn

c) error

c) abort

2.7.2 Semantics

A use-check is evaluated by the network linker by evaluating its boolean-expression. If the result is true, the use-check succeeds; otherwise the diagnostic is performed.

A diagnostic is performed by the network linker by evaluating its string-expression and then sending the result to the network linker log. If the diagnostic-severity is "warn", the diagnostic succeeds. If it is "error", the check fails and the result will be a faulty node network. If it is "abort", the check fails and the linker will immediately halt linkage of the current node and any nodes dependent on the current node; the result of linking will be an incomplete and faulty network.

3.0 LINKS

This section deals with the way nodes are interconnected using LINGO Links. The two types of links, running and memory, are defined in terms of their execution time actions.

3.1 SYNTAX

```
a)  link imp:          open pointy token, m link imp,
                          close pointy token;
                       r link imp;
                       export;
                       sump.
b)  m link imp:        m link id.
c)  r link imp:        r link id.
d)  export:            port token, export id.
e)  sump:              sump token.
f)  link ref:          open pointy token, m link ref,
                          close pointy token;
                       r link ref.
g)  m link ref:        port token, import id;
                       init value;
                       init value, simple node invocation,
                          link env option;
                       init value, m link id,
                          link inv option.
h)  r link ref:        port token, simple node invocation,
                          link env option;
                       r link id, link inv option.
i)  init value:        node id, node modifications pack option.
j)  link env:          environment token,
                          link type name list pack.
k)  link type name:    string expression.
```

Examples

a) <pdata>

a) data

a) port user

a) *

b) parms

c) matrix

d) port sysprint

e) *

f) <default, parms>

f) matrix

g) port sysin

g) blankline, randomdata (thing)

g) blankline, purple

h) randomdata (thing) env (spool)

i) blankline

i) line (/t = "1 Statistics of Deadly Quarrels")

j) env (spool)

k) "/sys/spool2"

k) spool

3.2 SEMANTICS

Links are used to carry data, in the form of records, from the export of one node to the import of another. Exports are always associated with link-imp's and imports with link-ref's. The record type of the data must be consistent with both the export record type (where it is derived) and the import record type (where it must be compatible). The definition of compatible is given in Section 5.4.

A particular link is defined by starting at any link-imp and drawing a line to the single link-ref that shares the same m-link-id (r-link-id). There may be any number of starting points that feed a common termination. If more than one m-link-imp (r-link-imp) shares a common termination, the group of links are called merging links. Link names are entirely local to a node network. Any reference to a non-local link must be done through a node network port.

For convenience, a link can be considered to have a name and a record-type. The link's name is the one used in a local node-network to place data in it and receive data from it. Its record-type is the same as the records that it carries. Merging links also have a single name and type.

When a record is placed on a link in some link statement, the record traverses the link and eventually arrives at the terminating end. Links always maintain records in a first-in first-out order. If records are already present on the link when a new record is input, the new record is added to the end of a queue at the termination. On a merged link records may arrive in an arbitrary order from the various sources before being placed the queue formed at the termination.

A link-imp may be a link, an export, or a sump. The link case was described in the previous paragraph. An export works in a similar way except the record is placed at the starting end of the external link connected to the nodes export that is bound to the referenced export-id. A sump works by discarding the record supplied.

Link-ref's are the terminating end of a link and are associated with a node's import. Both m-link's and r-link's operate in an identical manner; however, node invocation makes a distinction between them, and m-link-ref's must have an init-value specified unless they refer to an import. The init-value record is placed on the m-link before the network begins to operate; it is obtained by invoking the import-less node identified by its node-id and placing the resulting record on the m-link. The import-less initial value node must yield exactly one output per invocation on its single export.

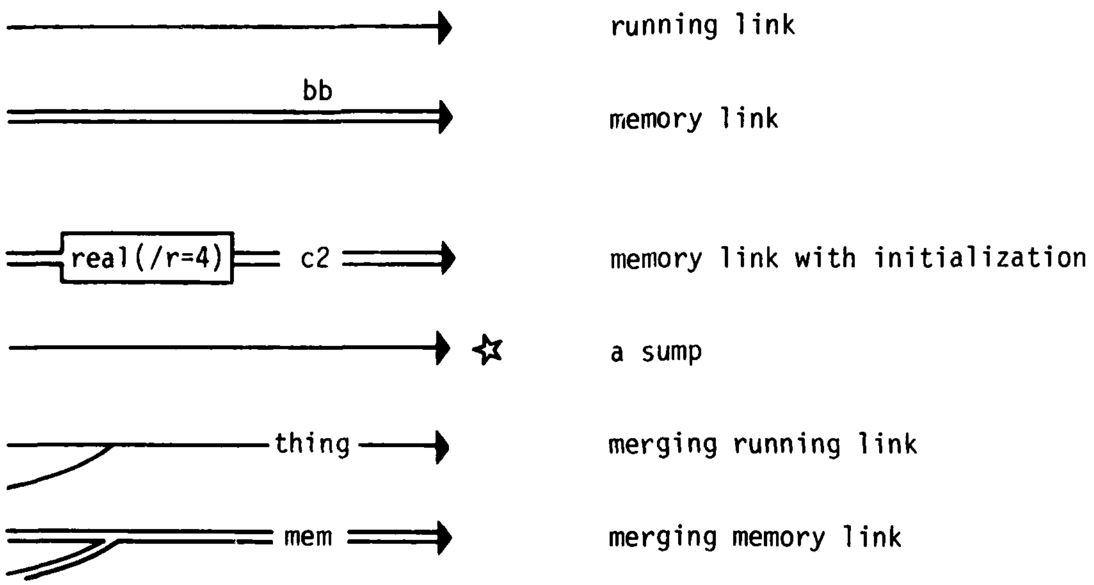
A network of r-link's that form a backwards loop is prohibited. A branch and merge system of running links will be handled by the performance analyzer and spools inserted automatically if necessary.

Besides records, running links also transmit end-of-file markers. An end-of-file is handled somewhat differently than a record. Memory links do not transmit end-of-file markers. Merged running links do not place the end-of-file marker at the end of the termination's queue of data until an end-of-file has been received from all the starting link-imps in the merge. A running link is active when it has records on it or when it is transmitting an end-of-file. A running link is inactive if the node has received the end-of-file marker and removed it from the running link.

The link-env-options provide implementation-dependent information. For example, the user might indicate a file name that will contain the data records. Thus, at the end of network execution the file will contain all the data that traversed the link.

3.3 GRAPHICAL FORM OF LINKS

A running link is represented by a single line and a memory link by two parallel lines. Merging links are represented by merging lines and a sump by an asterisk. If a link has a link-id, the identifier is written on or next to the link. Thus:



4.0 DATA PATHS: THE NETWORK ACTIONS

4.1 COPIES

4.1.1 Syntax

a) data path: copy; node invocation.

b) copy: copy token, r link ref, into token, link imp list.

Examples

b) copy a --> b, c, <d>, b, port e

b) copy source (input) --> <f>, link2, j, k

b) copy port m --> ml, m2

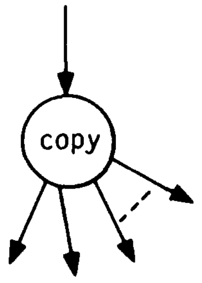
4.1.2 Semantics

The copy statement acts like a node that the user could build for himself. It is included for its convenience and generality. Whenever the copy statement is active, it awakens when a record is present on its r-link-ref. The data is read and copied to all the link-imp's in the limk-imp-list. The input must be a running link, but the outputs from the copy may be a mixture of running, memory, and exports. The type of each link-imp is the same as the type of the r-link ref. After making the copy, the statement goes to sleep.

An end-of-file is copied from the r-link-ref to each running and export link-imp. After the end-of-file is copied, the copy statement becomes inactive.

4.1.3 Graphical Form of Copies

A copy is represented by a circle with one input and any number of outputs. The circle is named "copy".



4.2 NODE INVOCATIONS

4.2.1 Syntax

a) node invocation: node id, arguments pack, into token, link imp list.

b) simple node invocation: node id, arguments pack.

c) arguments: link ref list option, node modifications option.

d) node modifications: slash token, node modifier list.

e) node modifier: modifier id, is token, static expression.

Examples

```
a)  random line (link_a, link_b) --> link_c, link_d
                                  {two inputs, two outputs}
a)  add (link_a, link_b) --> c {two inputs, one output}
b)  add (link_a, link_b)       {two inputs, one output}
c)  link_a, link_b
c)  link_a, <t, bit>/m = 14*i
d)  /m = 14*i, n = 27, t = "random"
e)  m = 14*i
e)  m = 27
```

4.2.2 Semantics

The network linker uses a node invocation to identify a particular node by employing the following algorithm:

1. Obtain input record-types for each import by examining its link-ref.
2. Obtain the node-id for the node to be linked.
3. Make the most local node-network the search space.
4. Find all node definitions in the search space which have a matching node-id and corresponding number of imports to the number entering this node.
5. For all node definitions found in step 4, find if the respective record-types are compatible with the formal-record-types given in the definition.

6. If only one node definition has compatible links, that node is selected and the search is terminated.
7. If more than one node definition has compatible links, a warning is issued, the first textual one is chosen, and the search is terminated.
8. If no node definition is found with compatible links, the encompassing node-network is made the search space and step 4 is taken.
9. If there are no more encompassing node-network's, a matching definition cannot be found and the linker issues an abort message. It should continue to link any other nodes that can be linked until there is nothing more to do.

When the proper node-def is found, the node-modifier-list from the arguments-pack is applied. This yields a set of values for the modifiers, each of which replaces its corresponding modifier in the proper node-network. Then the use-check's from the node-network of the node-def are evaluated to check the suitability of the node and issue appropriate messages. If the node passes every use-check, it is linked into the network.

The node-def includes the specification of one or more ports. The imports are linked by matching them to corresponding positions in the arguments-pack. The exports in a node invocation are matched to corresponding positions in the link-imp-list. The actual-record-type of each export, and thus each link-imp, is derived from the modifiers and the output record-type for each link. The number of exports of the node-def must match the number of link-imp's in the node invocation.

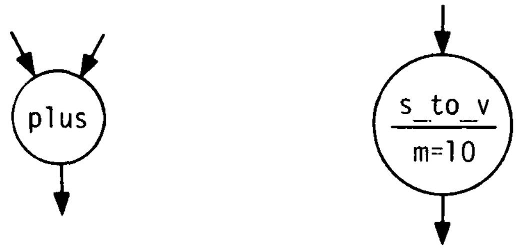
A simple-node-invocation has only a single export and it is matched with the one in the export-list of the node-def. The simple-node-invocation implies an unnamed running or memory link by the place it is used in the syntax. This running or memory link acts exactly the same way a named link would act.

When a node is active it may be awakened or invoked whenever all its active r-link-ref's connected to its arguments contain a record. To invoke or awaken a node means that the next record on all available link-ref's are transferred to the node's imports and the corresponding records are removed from the links that carried them. Thus, if nothing has arrived on a memory link (m-link-ref), no transfer to the corresponding import will be made and the node's import will retain the same value, thus "remembering" the last arrival.

An end-of-file that arrives on an r-link-ref changes the link from active to inactive. This may cause the node to awaken by removing from consideration one of the imports attached to a running link. When all r-link-ref's are inactive but the node is active, it is invoked or awakened for one last fling. After this final invocation, when the node falls asleep, an end-of-file is placed on each running link associated with an export and the node is made inactive.

4.2.3 Graphical Form of Nodes

A node is represented by a circle with imports as arrow inputs to the circle and exports as lines out. The name of the node is written in the circle. If modifiers are present they are written in the circle separated from the node name by a line. Thus:



5.0 RECORDS AND NETWORK DATA

Data that flows on LINGO network links is described in terms of length (number of bits) and usage (type information). The length is used to calculate buffer sizes and record lengths. The usage information allows mismatches between links and imports to be detected by the network linker. The type information is arbitrary and dependent on the node designer. This is reasonable because machine data types may differ in a network and only the node library designer has the proper perspective to choose types.

Data may be grouped into blocks by building arrays and structures of other data. Grouped data provides the form that describes the records used to carry information from node to node within a network defined by an application node library. Seldom will a node work on just a simple arithmetic or logical quantity because this would imply too low a level of programming for the network language to be effective. The notion record-type is never directly referred to in the other parts of the grammar and will never directly occur in a LINGO program because the grammatical notions of formal- and actual-type describe the language use of a record-type. However, by the time linkage has successfully completed, every link and associated port has a fully defined record-type which characterizes it.

Section 5.1 covers record-types and thus will never be used in parsing a LINGO source program. Section 5.2 covers formal-types, each of which describe the set of record-types an import will accept. Section 5.3 describes actual-types each of which describes a single record type which can be computed by the network linker.

5.1 BASIC ORGANIZATION OF RECORDS

5.1.1 Record Type

5.1.1.1 Syntax

a) record type: basic element; array data; structure data.

Examples

a) bit (16) type ("2s compl", "fixed")

a) dim (1:16) bit (1) type ("logical")

a) struct (bit (16) type ("logical"),
bit (32) type ("real", "IBM360"))

5.1.1.2 Semantics

The record-type describes the form of any record which traverses a link. The record is the logical information holding entity that can be communicated between nodes. Every record, basic-element, array-data, and structure data possesses the attribute of length based on the number of bits in all its subcomponents.

5.1.2 Basic Data

5.1.2.1 Syntax

a) basic element: bit token, length pack, basic attribute.

b) length: integer value.

c) basic attribute: type token,
machine format name list pack.

d) machine format name: string value.

Examples

a) bit (16) type ("2s compl", "fixed")

a) bit (64) type ("long real", "IBM370")

b) 16

c) type ("2s compl", "fixed")

d) "2s compl"

5.1.2.2 Semantics

The basic data is an arbitrary type, depending on the needs of the node library designer. Associated with each type is a length-pack giving the number of bits used to represent the data, as well as a list of machine-format's. The set of machine-format's is chosen by the node library designer.

5.1.3 Array Data

5.1.3.1 Syntax

a) array data: dimension token, bounds list pack, record type.

b) bounds: lower bound, colon token, upper bound.

c) lower bound: signed integer value.

d) upper bound: signed integer value.

Examples

a) dim (1:16) bit (16) type ("2s compl", "fixed")

b) 1:16

c) 1

d) 16

5.1.3.2 Semantics

Array data represents a series of values in a record in row-major order in which varying the right most subscript is varied the fastest.

Arrays are specified by preceding a record-type with "dim (lb_1: ub_1, lb_2: ub_2, lb_3: ub_3, . . . lb_n: ub_n)". This indicates an n-dimensional array with successive pairs of lower-bound (lb) and upper-bound (ub). The lower-bound and upper-bound may be parameterized in formal-types to allow the network language processor to check dimensionally between input ports and to derive the size of output arrays. All size descriptors for arrays must be able to be established prior to executing the network.

The array size of both input and output ports is accessible from a primitive node by a modifier enquiry so that of any sized array can be properly processed by the node. The value returned by the modifier will not change during the execution of a particular node, although it may differ if another instance of the same node is used in another place in the network.

5.1.4 Structures

5.1.4.1 Syntax

a) structure data: structure token, string value,
record type list pack.

Examples

a) struct ""(bit (16) type ("logical"), bit (32) type
("real", "IBM360"))

a) struct "string" (bit (16) type ("unsigned", "fixed"),
dim (1:256) bit (8) type ("char"))

5.1.4.2 Semantics

Structure data represents a series of values on a record in the same order of the data in the compound-data-list. The string-value allows structure types that would otherwise be the same to be distinguishable.

5.2 ACTUAL DATA DESCRIPTIONS

5.2.1 Syntax

a) actual type: actual record type;
actual parameterized type.

b) actual record type: actual basic element;
actual array data;
actual structure data;
anytype id.

c) actual basis element: bit token, actual length pack,
actual basic attribute.

d) actual length: integer expression

e) actual basic attribute: type token,
actual machine format name list pack.

f) actual machine format: string expression

g) actual array data: dimension token,
actual bounds list pack,
actual record type.

h) actual bounds: actual lower bound, colon token, actual upper bound.

i) actual lower bound: integer expression.

j) actual upper bound: integer expression.

k) actual structure data: structure token, actual tag, actual type list pack.

l) actual tag: string expression.

Examples

a) vector(n)

a) struct "flex" (integer, dim (l:n) t)

b) struct "flex" (integer, dim (l:n) t)

c) bit(n) type (twos_compl, fixed)

d) n

e) type (twos_compl, fixed)

f) twos_compl

f) "2s compl"

g) dim (l:n) t

h) l:n

i) l

j) n

k) struct "flex" (integer, dim (l:n) t)

l) "flex"

5.2.2 Semantics

The actual-type is used to define the record-type of an export in terms of link time expressions, which can be in terms of modifiers set at the imports, in the library, by the user employing the node, or by default. The result of evaluating an actual-record-type is the record-type obtained by evaluating every expression and expanding every parameterized-type within the actual-record-type.

5.3 FORMAL DATA DESCRIPTIONS

5.3.1 Syntax

a)	formal type:	formal record type; formal parameterized type.

b)	formal record type:	formal basic element;
		formal array data;
		formal structure data;
		anytype attribute.

c)	anytype attribute:	anytype token, anytype id.
d)	formal basic element:	bit token, formal length pack, formal basic attribute.
e)	formal length:	integer expression; formal modifier.
f)	formal basic attribute:	type token, formal machine format name list pack.
g)	formal machine format name:	string expression; formal modifier.
h)	formal array data:	dimension token, formal bounds list pack, formal type.
i)	formal bounds:	formal lower bound, colon token, formal upper bound.
j)	formal lower bound:	integer expression; formal modifier.
k)	formal upper bound:	integer expression; formal modifier.
l)	formal structure data:	structure token, formal tag, formal type list pack.
m)	formal tag:	string expression; formal modifier.
n)	formal modifier:	modifier token, modifier id.

Examples

```
a) vector (mod n)
a) struct "flex" (integer, dim (1: mod n) anytype t)
b) struct "flex" (integer, dim (1: mod n) anytype t)
c) anytype t
d) bit (mod n) type (twos_compl, fixed)
e) mod n
f) type (twos_compl, fixed)
g) fixed
h) dim (1: mod n) anytype t
i) 1: mod n
j) 1
k) mod r
l) struct "flex" (integer, dim (1: mod n) anytype t)
m) "flex"
n) mod n
```

5.3.2 Semantics

Formal-types are used to describe imports in terms of modifier and anytype-id's so that a class of record-type's can be processed. A formal-type can define one or more modifier-id's, each of which assumes a value whenever the mode is linked.

5.4 COMPATIBILITY OF RECORD TYPES

Exports from a linked node can be characterized by the grammar notion record-type. This implies that each piece of basic data has a given number of bits associated with it and that it defines the structure that relates the parts. Imports to a node are defined in a parameterized fashion and are described by a formal-record-type. An algorithm must exist that determines if the record-type on an input link is compatible with the formal-record-type characteristics of an import.

The algorithm for compatibility operates as follows:

1. Expand any parameterized-type's until all that remains is a formal-record type. This description will contain structure, specific numbers, modifier-ids, and anytype items.
2. Replace all formal-modifier's with a hash mark, "#", and all anytype-attribute's with a dollar sign, "$". Compress all blanks and comments from both data descriptions.
3. Match descriptions character-for-character starting at the left-most item. If both descriptions match, they are compatible. If the first character in which they do not match is a "#" in the modified formal record description, skip the corresponding string or number of the record description. The "#" should match the first character of the string or number. The modifier-id that the "#" represents assumes the value of the matched item. If the first character in which they do not match is a "$" in the modified formal record description, skip the corresponding basic-element, array-data, or structure-data component of the record description. The "$" should match the first character in such a component. The ayntype id that the "$" represents assumes the type of the matched item. After skipping, continue to apply rule 3. If there is a character mismatch that does not correspond to either the "#" case or the "$" case, the record-types are not compatible.

5.5 PARAMETERIZED TYPES

5.5.1 Syntax

a) actual parameterized type:

    record type id,
        actual type parameter list pack option.

b) actual type parameter:

    static expression.

c) formal parameterized type:

    record type id,
        formal type parameter list pack option.

d) formal type parameter:

    formal modifier; static expression.

Examples

a) matrix (10, 20)

a) vector (n+1)

a) integer

b) 10

b) n+1

c) matrix (10, 20)

c) matrix (mod m, mod n)

c) vector (n+1)

c) integer

d) 10

d) n+1

d) mod m

5.5.2 Semantics

A parameterized-type yields a type by substituting each of its type-parameter's for its corresponding modifier-id in the string-expression of the record-type-def associated with the record-type-id. The number of type-parameter's must match the number of modifier-id's. If the resulting string contains parameterized-type's, those types must be expanded according to the same rules. Note that a formal-parameterized-type always yield a formal-type and an actual-parameterized-type always yields an actual-type.

6.0 COMPUTATIONS AT LINK TIME

6.1 STATIC EXPRESSIONS

6.1.1 Syntax

a) static expression: operator expr; function expr; substring expr; parenthesized expr; if expr; primitive expr.

b) boolean expression: static expression.

c) integer expression: static expression.

d) string expression: static expression.

6.1.2 Semantics

Every static-expression yields a character string when evaluated. The details for evaluation are given in the semantics of each kind of expression later in this section.

There are two classes of character strings that are distinguished from the general form. The first class in Boolean. Only two strings are Boolean: "TRUE" and "FALSE". The second class is integer. Integers ar strings of the following form: an optional minus sign, concatenated with an optional sequence of digits the first of which is nonzero, concatenated with a single digit.

A boolean-expression, when evaluated, must yield a Boolean value. An integer-expression, when evaluated, must yield an integer value. If these restrictions are not met, an error message will be issued by the network linker.

6.2 OPERATOR EXPRESSIONS

6.2.1 Syntax

a) operator expr: unary expr;
binary expr;
optimized expr.

b) unary expr: unary operator, static expression.

c) binary expr: static expression, binary operator, static expression.

d) unary operator: not token; plus token;
minus token; length token.

e) binary operator: eq token; ne token; lt token; le token;
gt token; ge token; eqs token;
nes token; lts token; les token;
gts token; ges token; plus token;
minus token; times token;
divided by token; exponentiation
token; concatenation token;
repetition token; index token;
scan token; verify token.

f) optimized expr: boolean expression, and token,
boolean expression;
boolean expression, or token,
boolean expression

Examples

b) - (3+4)

b) #s

c) a*b*c-d

c) s#*(3+d)

d) -

e) #*

f) (a<b) & (c<d)

6.2.2 Semantics

If an operator-expr contains several operators, the network linker will use operator precedence to determine which operations are performed first.

The highest priority operator is performed first. If two operators of equal priority appear in an operator-expr, the leftmost is performed first. Priorities appear in Table A-2.

TABLE A-2 OPERATORS

TOKEN	SYMBOL	CLASS OF 1st ARGUMENT	CLASS OF 2nd ARGUMENT	CLASS OF RESULT	PRIORITY	SEMANTICS
not	~	Boolean	-	Boolean	10	Boolean negation
plus	+	integer	-	integer	10	No effect
minus	-	integer	-	integer	10	Integer negation
length	#	string	-	integer	10	The length of the string in characters
and	&	Boolean	Boolean	Boolean	2	Boolean AND (optimized operator)
or	!	Boolean	Boolean	Boolean	1	Boolean OR (optimized operator)
eq	=	integer	integer	Boolean	3	Integer comparison operators
ne	=					
lt	<					
le	<=					
gt	>					
gr	>=					

TABLE A-2 OPERATORS (continued)

TOKEN	SYMBOL	CLASS OF 1st ARGUMENT	CLASS OF 2nd ARGUMENT	CLASS OF RESULT	PRIORITY	SEMANTICS
eqs	=*	string	string	Boolean	3	String comparison operators. A string is less than another string if it alphabetically precedes it. The collating sequence is used to determine alphabetic order. If strings of unequal length are to be compared then the shorter is considered to be extended on the right by "characters" which precede every character in the collating sequence.
nes	¬=*					
lts	<*					
les	<=*					
gts	>*					
ges	>=*					
plus	+	integer	integer	integer	7	Integer addition
minus	-	integer	integer	integer	7	Integer subtraction
times	*	integer	integer	integer	8	Integer multiplication
divided by	%	integer	integer	integer	8	Integer division (result truncated)
exponentiation	**	integer	integer	integer	9	Exponentiation (result truncated)
concatenation	#	string	string	string	5	String concatenation

TABLE A-2 OPERATORS (continued)

TOKEN	SYMBOL	CLASS OF 1st ARGUMENT	CLASS OF 2nd ARGUMENT	CLASS OF RESULT	PRIORITY	SEMANTICS
repetition	#*	string	integer	string	6	String repetition: s##0="" s##1=s s##2=s#s s##3=s#s#s ⋮
index	::	string	string	integer	4	The index of the first position of where the second argument appears in the first. If it does not appear, then 1 plus the length of the first. E.g., "BABE"::"AB"=2, "BABE"::"EB"=5.
scan	#:	string	string	integer	4	The index of the first position of where a character in the second argument appears in the first. If it does not appear, then 1 plus the length of the first. E.g., "BABE"#:"AB"=1, "BABE"#:"CD"=5.

TABLE A-2 OPERATORS (continued)

TOKEN	SYMBOL	CLASS OF 1st ARGUMENT	CLASS OF 2nd ARGUMENT	CLASS OF RESULT	PRIORITY	SEMANTICS
verify	∿:	string	string	integer	4	The index of the first position of where a character not in the second argument appears in the first. If all characters in the first are in the second, then 1 plus the length of the first. E.G., "BABE"∿: "AB"=4, "BABE"∿: "ABCDE"=5

A unary-expr is evaluated by the network linker by evaluating its static-expression, performing the unary operation on the static-expression's result. A list of unary-operator's and their meaning can be fo:.d in Table A-2.

A binary-expr is evaluated by the network linker by evaluating its two static-expression's, performing the binary operation on the results, and yielding the result of the binary operation. A list of binary-operator's and their meanings can be found in Table A-3.

An optimized-expr is evaluated by the network linker by evaluating its first boolean-expression. If the result is such that the result of the optimized-expr is known, the second static-expression is not evaluated; otherwise the second static-expression is evaluated and the operation applied as in normal binary-expr's.

In any operator-expr, the class of the operands must be the class expected by the operator. If this restriction is not met, the network linker will issue an error message. Expected classes are given in Table A-2.

6.3 FUNCTION EXPRESSIONS

6.3.1 Syntax

a) function expr: function id, static expression list pack.

Examples

a) min (a, b, c, d)

a) abs (c+2)

a) isinteger (e)

a) evaluate ("a+b")

6.3.2 Semantics

A function-expr is evaluated by the network linker by evaluating its static-expressions, performing the function on the respective results, and yielding

TABLE A-3 FUNCTIONS

NAME	CLASS OF ARGUMENTS (1st, 2nd, ...)	CLASS OF RESULT	SEMANTICS
min	integer, integer, ...	integer	Accepts any number of integer arguments and returns their minimum
max	integer, integer, ...	integer	Accepts any number of integer arguments and returns their maximum
mins	string, string, ...	string	Accepts any number of string arguments and returns their minimum (in terms of the <* operator - see Table 6-2)
maxs	string, string, ...	string	Accepts any number of string arguments and returns their maximum (in terms of the <* operator - see Table 2-2
abs	integer	integer	Absolute value
sign	integer	integer	Sign (-1 if negative, +1 if positive, 0 if zero)
odd	integer	Boolean	True if argument is odd
even	integer	Boolean	True if argument is even
mod	integer, integer	integer	Integer modulus or remainder
isboolean	string	Boolean	"TRUE" if the string is a Boolean, "FALSE" otherwise
isinteger	string	Boolean	"TRUE" if the string is an integer, "FALSE" otherwise
bint	Boolean	integer	"1" if the Boolean is "TRUE", "0" if "FALSE"
evaluate	string	string	The result of evaluating the given string in the current environment

the result of the function. A list of functions and their arguments can be found in Table A-3. LINGO does not provide facilities for defining new functions or operators.

If any function-expr, the class of the argument(s) must be the class expected by the function, otherwise the network linker will issue an error message. Expected classes are given in Table A-3.

6.4 SUBSTRING EXPRESSIONS

6.4.1 Syntax

a)	substring expr:	non operator expr, sub token, substring chooser, bus token.
b)	substring chooser:	character chooser; string chooser.
c)	character chooser:	static expression.
d)	string chooser:	start pos option, colon token, end pos option.

e)	non operator expr:	function expr; substring expr;
		parenthesized expr; if expr;
		primitive expr.

f)	start pos:	static expression.
g)	end pos:	static expression.

Examples

a) s[3:5]

a) (s#"abcd") [:5]

a) s[n]

c) n

d) 3:5

e) s

e) (s#"abcd")

f) 3

g) 5

6.4.2 Semantics

A substring-expr is evaluated by the network linker by evaluating its non-operator-expression, yielding a string S; and by evaluating its substring-chooser, yielding a chooser C.

If C is a single integer i, the result of the substring-expr is a string consisting of a single character, the ith in S. If i is less than one or greater than the length of S, an error occurs.

If C is a pair of integers, i and j, the result of the substring-expr is a string consisting of the ith, (i+1)st, ..., (j-1)st, jth characters, respectively, in S. It is required that either i > j (in which case the result of the substring-expr is the null string) or that both i and j fall between 1 and the length of S, inclusive; otherwise an error occurs.

A string-chooser is evaluated by the network linker by evaluating its two pos-options, each of whcch must yield an integer. If the start-pos-option is empty, "1" is assumed in its place; if the end-pos-option is empty, the length of the string yielded by the non-operator-expr of the substring-expr containing the string-chooser is assumed in its place. The result of the string-chooser is a pair of integers representing the result of their respective expressions.

6.5 PARENTHESIZED EXPRESSIONS

6.5.1 Syntax

a) parenthesized expr: static expression pack.

Examples

a) (a+b)

a) (s# "abcd")

6.5.2 Semantics

A parenthesized-expr is evaluated by the network linker by evaluating its static-expression and yielding its result. Parentheses are used to alter the default order of the evaluation established by operator priorities, as given in Section 6.2.

6.6 IF EXPRESSIONS

6.6.1 Syntax

```
a)  if expr:               if token, if body, fi token.
b)  if body:               if section, then token, then section,
                              else body.
c)  else body:             else token, else section;
                           elif token if body.
d)  if section:            boolean expression.
e)  then section:          static expression.
f)  else section:          static expression.
```

Examples

```
a)  if a<b then a#b else b#a fi
a)  if a<b then -1 elif a+b then 0 else +1 fi
b)  a<b then a#b else b#a
b)  a<b then -1 elif a=b then 0 else +1
c)  else b#a
c)  elif a=b then o else +1
d)  a<b
e)  a#b
f)  b#a
```

6.6.2 Semantics

An if-expr is evaluated by the network linker by evaluating its if-body.

An if-body is evaluated by the network linker by evaluating its if-section. If the result is "TRUE" the value of the if-body is the result of evaluating the then-section; otherwise, it is the result of evaluating the else-body.

If the else-body begins with an else-token, its value is determined by evaluating its else-section; otherwise, its value is determined by evaluating its if-body.

6.7 PRIMITIVE EXPRESSIONS

6.7.1 Syntax

a) primitive expr: value id; modifier id;
value; length expr.

b) value: boolean value; integer value;
string value.

c) boolean value: true token; false token.

d) integer value: digit NUM token,
digit NUM symbol sequence option.

e) string value: open quote token, character sequence option,
close quote symbol.

f) signed integer value: sign, digit NUM symbol sequence;
integer value.

g) sign: plus token; minus token.

h) length expr: length token, anytype id.

Examples

a) max_int

a) n

a) 39

a) length t

c) true

d) 39

e) "daintiness of ear/To check time broke in a disordered string"

e) "" {empty string}

f) -39

g) -

h) length t

6.7.2 Semantics

A value-id is evaluated by the network linker by obtaining the value associated with the identifier. If no such identifier has a scope that includes the value-id (i.e., the use of an identifier without a declaration), or if two identifiers have such scopes (i.e., a doubly-defined identifier within the same node network), the network linker will issue an error message.

A modifier-id is evaluated in the same way as a value-id. An additional error that can occur with a modifier-id is the use of a modifier-id that did not have a default value and was never associated with a value.

When evaluated, a length-expr yields the size, in bits, of the record type represented by its anytype-id.

A value is evaluated according to its type. If it is a boolean-value, the result is "TRUE" for a true-token and "FALSE" for a false-token; if it is an integer-value, the result is the string that represents the integer, as in Section 6.1.1; if it is a string-value, the result is the represented string. An open- or close-quote-symbol is represented within a string-value by two such symbols in succession.

7.0 LINGO'S LITTLE THINGS

This section deals with comments, identifiers, basic tokens, and the LINGO character set. In the preceeding sections the symbols which may be preceeded by comments were indicated by the suffix "token".

7.1 COMMENTS

7.1.1 Syntax

a) NOTION token: artwork sequence option, NOTION symbol.
b) artwork: blank symbol; comment;
  typographical display feature.
c) typographical display feature:
  newline symbol; tab symbol;
  new page symbol; vertical tab symbol;
  other display feature.
d) comment: open comment symbol,
    character glyph sequence option,
    close comment symbol.

7.1.2 Semantics

The NOTION-token rule can be considered a model rule that exists for any grammatical notion ending in "token". The artwork represents blanks, comments, and typographical display features, all of which are used to beautify LINGO networks for a reader. There is no definition given here for other-display, feature; an implementor may choose what else to ignore; however, it may not be another symbol defined in this report. None of the character-glyphs within a comment may be an open-comment-symbol or a close-comment-symbol.

7.2 IDENTIFIERS

7.2.1 Syntax

a) NOTION id: tag.
b) tag: letter ALPHA token, alphanumeric sequence option.

c) alphanumeric: letter ALPHA symbol; digit NUM symbol; underline symbol.

7.2.2 Semantics

The NOTION-id rule can be considered a model rule for any grammatical notion ending in "id". A tag can be preceded by blanks and comments but it cannot have internal blanks and it must occur on a single line.

7.3 PACK AND OPTION

7.3.1 Syntax

a) NOTION pack: open parenthesis token, NOTION, close parenthesis token.

b) NOTION option: NOTION; EMPTY.

c) EMPTY:: .

7.3.2 Semantics

The NOTION-pack rule is a model rule for any grammatical notion ending in "pack". It places parenthesis around the notion. The NOTION-option acts similarly, providing a rule to make the notion optional.

7.4 LISTS, SEQUENCES, AND GROUPS

/.4.1 Syntax

a) NOTION list: NOTION;
NOTION, comma token, NOTION list.

b) NOTION group: NOTION;
NOTION, with token, NOTION group.

c) NOTION sequence: NOTION;
NOTION, NOTION sequence.

7.4.2 Semantics

These are three model rules that define how lists of items separated by commas, semicolons, etc., are derived for notions ending in the appropriate keyword.

7.5 LINGO'S SYNTACTIC SYMBOLS

Table A-4 lists the LINGO symbols and a proposed typographical representation for each one. The lists are organized into the basic areas of the language: structural symbols, operator symbols, and declarative symbols. When LINGO programs are written or printed, the underline beneath the symbol is recommended so that keywords are easily recognized. However, when LINGO programs are input on a terminal, the underline can be ignored and the LINGO lexer will recognize them by using the usual keyword scheme. When keywords are input, no blanks are allowed within the word. The character set has been carefully designed to stay within the ASCII 128 character set.

7.6 THE LINGO CHARACTER SET

Table A-5 lists the LINGO symbols that constitute the basic character set used for identifiers and numbers.

7.6.1 Syntax

```
      a)  character:       letter ALPHA symbol;
                           digit NUM symbol;
                           other character symbol;
                           open quote symbol, open quote symbol;
                           close quote symbol, close quote symbol.
```

7.7 METAPRODUCTION RULES FOR GENERAL FORMS

7.7.1 Syntax

```
      a)  ALPHA::           a; b; c; d; e; f; g; h; i; j; k; l;
                            m; n; o; p; q; r; s; t; u; v; w;
                            x; y; z.
```

STRUCTURAL SYMBOLS		DECLARATIVE SYMBOLS	
Name	Representation	Name	Representation
import symbol	import	node symbol	node
export symbol	export	record symbol	record
environment symbol	env	definition symbol	def
begin symbol	begin	port symbol	port
end symbol	end	modifier symbol	mod
ms symbol	ms	library symbol	lib
output symbol	out	unless symbol	unless
invocation symbol	inv	warn symbol	warn
with symbol	;	error symbol	error
comma symbol	,	abort symbol	abort
copy symbol	copy	bit symbol	bit
into symbol	-->	type symbol	type
open pointy symbol	<	dimension symbol	dim
close pointy symbol	>	structure symbol	struct
slash symbol	/	anytype symbol	anytype
sump symbol	*	colon symbol	:
open parenthesis symbol	(	is symbol	=
close parenthesis symbol	)		
open comment symbol	¦		
close comment symbol	¦		

OPERATOR SYMBOLS

Name	Representation	Name	Representation
not symbol	~	close quote symbol	"
plus symbol	+	times symbol	*
minus symbol	-	divided by symbol	%
length symbol	#	exponentiation symbol	**
eq symbol	=	repetition symbol	#*
ne symbol	~=	index symbol	::
lt symbol	<	scan symbol	#:
le symbol	<=	verify symbol	~:
gt symbol	>	and symbol	&
ge symbol	>=	or symbol	!
eqs symbol	=*	sub symbol	[
nes symbol	~=*	bus symbol	]
lts symbol	<*	open parenthesis symbol	(
les symbol	<=*	close parenthesis symbol	)
gts symbol	>*	if symbol	if
ges symbol	>=*	then symbol	then
true symbol	true	else symbol	else
false symbol	false	elif symbol	elif
open quote symbol	"	fi symbol	fi

TABLE A-5 LINGO ALPHABET

Name	Representation	Name	Representation
letter a symbol	a	letter n symbol	n
letter b symbol	b	letter o symbol	o
letter c symbol	c	letter p symbol	p
letter d symbol	d	letter q symbol	q
letter e symbol	e	letter r symbol	r
letter f symbol	f	letter s symbol	s
letter g symbol	g	letter t symbol	t
letter h symbol	h	letter u symbol	u
letter i symbol	i	letter v symbol	v
letter j symbol	j	letter w symbol	w
letter k symbol	k	letter x symbol	x
letter l symbol	l	letter y symbol	y
letter m symbol	m	letter z symbol	z
digit zero symbol	0	digit 5 symbol	5
digit one symbol	1	digit 6 symbol	6
digit two symbol	2	digit 7 symbol	7
digit three symbol	3	digit 8 symbol	8
digit four symbol	4	digit 9 symbol	9
underline symbol	_		

b) NOTION:: ALPHA;
NOTION, ALPHA.

c) EMPTY:: .

d) NUM:: zero; one; two; three; four; five; six;
seven; eight; nine.

7.7.2 Semantics

These rules create an independent context-free grammar which, when the terminal notions (consisting of all small letters) are used to replace the large notion in other rules in this book, create a grammar rule. Thus:

NOTION option: NOTION; EMPTY.

If by using the above meta-grammar one derived "program" from the rule for "NOTION" and "" from "EMPTY", the replacement would create a new grammar rule:

program option: program; .

This formally defines the syntactic suffixes used previously.

8.0 ROLE OF THE APPLICATION LIBRARY DESIGNER

LINGO provides a method for linking preconstructed programs together into a useful network. Reliable behavior of such a network can only be predicted if the application node library designer has given an adequate, honest description of the limitations, behavior, and performance of each of the nodes used. This means the data types that nodes manipulate should be carefully designed to be neither too narrow nor too broad: too narrow and they overly restrict the user's ability to perform experiments; too broad and they will not protect the user from mistakenly misusing nodes. Therefore, to a large extent the effectiveness with which LINGO can detect user mistakes depends on information contained in the node libraries employed.

This section is devoted to informally examining the characteristics of nodes that will operate well in a LINGO network. In particular it will show what node behavior should be avoided in order to effectively use LINGO constructs. It is suggested that the reader read this section in conjunction with the example library presented in Section 9 to see the application of the principles discussed here.

8.1 PRIMITIVE NODE ACTIONS

A program associated with a node becomes "active" when a network that references the node is activated by the operating system. This might be thought of as the loading process in a conventional operating system. Unless a node is active, the program cannot perform computations.

A program is normally invoked when it has data to process. If the node associated with a program is reentrant, it could be simultaneously active with itself. Each activation would be processing successive input quantities. However, the output buffers associated with the invocations will be sequenced so that exports emerge in the intended order. This is done to preserve data synchronization in the network, a fundamental LINGO concept.

While processing data, a node cannot unexpectedly request additional data from a link. A node program should be conceived in terms of processing one set of input data which provides corresponding output(s). In the parlance of LINGO this is referred to as a single invocation.

A special invocation may occur when an end-of-file arrives at an import. The end-of-file deactivates the port and may cause invocation of the node if records have previously arrived on all remaining runnina links. Data can continue to arrive at these other ports, which will continue to activate the node. Therefore, for every invocation the node program could check a status flag associated with each import to determine if new data has arrived for this invocation. If the node cannot operate successfully with partial inputs, the node should issue an error message and terminate. It is anticipated that in such situations the most common action will be for the node to use the last piece of data that had arrived on the deactivated ports.

When the last end-of-file arrives, the one which deactivates the final-remaining import connected to a running link, the node is invoked for end-of-file processing. Many nodes will ignore this invocation and just terminate. Other nodes may compute summary information and export it on a special link before they terminate.

A program may have static storage that successive imports will modify. For example, a summation node could have one import and two exports. Normally the data at the import would be summed and passed to the export. When an end-of-file arrived at the import the summation node would export the total on the other link.

A program may also make use of secondary storage. For example, a sort node may have one import and one export. Unordered data would flow in the input and be placed into a secondary storage. When an end-of-file arrived at the import, the node would sort the data and export of all it on its output link.

A LINGO node-program can be written in any language for which an implementor has provided interface subroutines. These interface subroutines provide access to import/export links and status information. Thus, Fortran, Assembler, Cobol, PL/I, Algol or some other algorithmic language would be suitable for primitive nodes. It is recommended that any program that is destined to become a LINGO node include as many LINGO node declarations within comments in the program code as it is possible for the application library designer to provide.

8.2 INPUT/OUTPUT PROPERTIES

Nodes have characteristic patterns of behavior that describe the way outputs occur with respect to invocations. The best behavior, from the point of view of the network linker, is for every invocation to result in a fixed number of outputs for each export. The amount of output form export to export may differ, but for any one port the amount output in successive invocations must be fixed. This is called geared behavior because the ratio of invocation to exports on a given link is calculable at link time. With geared behavior, the buffer requirements are completely known in advance of execution of the network.

A less desirable behavior pattern is switched behavior. With switched behavior, some subset of the exports are alternatives. For any given invocation, output will occur on only one export and the other exports belonging to the subset will be unused. With a switched behavior, fairly good estimates of buffer requirements are possible.

The worst type of node behavior is probabilistic behavior. This occurs where the node varies the amount of data or which exports are used from invocation to invocation. In this case, buffer space for the worst case must be allocated to the node.

In designing an application library, nodes that exhibit probabilistic behavior should be avoided because the resources necessary to support them are great and performance prediction for succeeding nodes in the network is difficult. It should be noted that in current systems of programs most things that vary performance are due to run time dynamic resource allocation because link time semi-dynamic resource allocation is not available as it is in LINGO.

8.3 DATA TYPE USAGE

The data types belonging to links and ports are representative of concepts the eventual user will use to interconnect programs. The application library designer must conceive of these types in an implementation independent way so that he maximizes the effectiveness of the node library. However, LINGO requires that the application node designer specify the form of the data in machine-oriented detail. Therefore, it is the node designer's duty to translate the abstractions that will be employed by users into the concrete machine-dependent implementation details employed by the network executor. For example, if a system requires many parallel streams of data to be processed, each stream could be associated with a link type and the user would have to duplicate the network for each such stream he wished to process. Alternatively, the type could include a tag to represent which stream it originated from. In that case, one link would carry all the data and a modification of the number of streams to process would involve changing only the source node. If the parallel streams must be synchronized, some thought must be given to the conventions used by the source node.

It is considerations such as these which drive the choice of data types that an application library will support. That choice of data types is the most important decision an applications node designer will make because what the nodes are capable of doing is by-in-large determined by the form of the information they receive. In addition, the network linker chooses between nodes of the same name on the basis of the import link types. Well-designed data types will make the library very easy to use and facilitate modification of a network to perform similar functions.

From this point of view, a LINGO application library should be a well thought out amalgamation of data types, processing nodes, and predefined entities that make it easy for a user to construct networks that accomplish some needed computation. In constructing the network a user should find it impossible to mistakenly connect incompatible nodes and should come to expect timely, reasonable diagnostics even for subtle misuses of nodes. It is the application library designer who has the responsibility to live up to these goals. LINGO provides the facilities that allows him to accomplish them.

8.4 DATA TYPE FACILITIES IN LINGO

LINGO requires great detail in the specification of data types that characterize links. This detail was provided to prevent mistaken interconnection of incompatible links. It is not expected that a normal user will wish to specify each type in such detail. The mechanism that frees the user of this burden is the use of record type definitions.

The choice of these definitions is closely related to the actual choice of record types as described in Section 8.3. The importance of the definitions lies in the decision of which parts of the record type should be parameterized. With some types, such as

```
        record complex = "struct ""compl"" (real, real)";
```

or

```
        record real = "bit(64) type (""double"", ""IBM360"")";
```

there should be no parameterization because nodes that operate on such types cannot be easily parameterized to operate on similar records. However, some records, such as

```
        record vector(n) = "dim (1:" # n # ") real";
```

are more naturally able to be parameterized because a node program can easily be coded to handle arrays of varying sizes.

In short, in designing record type definitions an applications node designer must carefully choose between the desires of convenience and implementability (which favor fewer parameters) and flexibility (which favors more).

8.5 TOPOLOGICAL RESTRICTIONS

In designing an application library the LINGO constraint that forces flow of data in a single consistent direction on running links within a network may seem unduly restrictive. However, the restriction can often be eliminated using side effects within a single node. For example, a summation node could replace an add node which would sum data with a partial sum record running counter to the direction of the network.

It is possible to route data backward via a memory link. However, the receiving node must be prepared to synchronize the data by using local storage for data on the running link. The data transmitted on the memory link should have a sequence count or an alternating bit to allow determination of a new block arrival. This places the burden of storage and control within the node and causes performance estimates and output behavior to be probabilistic.

8.6 LIBRARY DESIGN PHILOSOPHY

The theme of this section has been that the application library designer has not only the responsibility to provide maximum security for the user, but also to provide for flexibility and generality of node design. Since the application library designer must provide performance estimates and algorithmic code for nodes, an application library design is tied to the performance of the nodes on specific machines.

In doing the library design it is wise to consider the machine-dependent aspects of the design and localize them into a separate series of type and value declarations upon which the remainder of the library can be built. Then the work required to transport an application library will center on respecifying the base data types along with recompiling the source code and validating the performance measures.

It should be pointed out that good diagnostics are the library designer's responsibility. If a user employs link types that do not match a node, the network linker will issue a "node not found" message. This will probably be somewhat frustrating for the user. If the library designer has anticipated common errors of this type he will provide a dummy diagnostic node, which will give the user a reasonable diagnostic message and point out a course of action that will correct the situation.

9.0 AN EXAMPLE APPLICATIONS LIBRARY

9.1 A MATRIX MANIPULATION LIBRARY

This section gives the specifications for a LINGO node library that performs matrix manipulations. This example was chosen for its clarity rather than for its complexity. Matrix algebra is fairly standardized and operations in this section are taken directly from the matrix subroutines given in a computer science textbook.* Section 9.2 will give an example of how a user would employ this library.

9.1.1 Choice of Data Types and Defaults

As was noted earlier, the crucial decision in the design of a node library is the choice of data types to be used. For simplicity it will be assumed that all the programs are running on a PDP-10 and all are handling 72-bit floating point numbers. For this reason the basic entry in a vector or matrix will be called a "scalar" where "scalar" is a record-type-id which expands into the appropriate definition. Once this is done the definitions for "vector" and "matrix" are straightforward.

```
record scalar      = "bit(72) type (""PDP-10"", ""float"")";
record vector(n)   = "dim(1:"#n#") scalar";
record matrix(m,n) = "dim(1:"#m#",1:"#n#") scalar";
```

Thus, for example, if a user writes the parameterized-type

```
matrix(15,25)
```

it will be expanded to

```
dim(1:15, 1:25) bit(72) type ("PDP-10", "float")
```

which is what was desired.

*Lindsey, C.H. and van der Meulen, S.G., Informal Introduction to ALGOL 68. London, North-Holland Publishing Company, 1977, revised edition.

Some routines (e.g., matrix transposition) will operate correctly on any 72-bit quantity, regardless of whether it is a floating point number. For these routines a slightly more general version of vector and matrix can be defined:

```
record row(n,t)     = "dim(1:"#n#")"#t;
record table(m,n,t) = "dim(1:"#m#",1:"#n#")"#t;
```

where t can be any type. Within each such routine a use-check will insure that the type is 72 bits large.

Once the data types are chosen, some thought should be given as to what defaults the user would like to use at each node. This cannot really be determined until the library has been designed, but two things that would be nice to specify when the library is included is the size of vectors (n) and matrices (m by n). These will be determined by the library modifiers "default_m" and "default_n" that a user can specify when the library is included.

The choices made in this section are summarized in the text below which is the first part of the matrix manipulation library.

```
mod default_m, default_n;
def record scalar      = "bit(72) type(""PDP-10"", ""float"")";
    record vector(n)   = "dim(1:"#n#") scalar";
    record matrix(m,n) = "dim(1:"#m#", 1:"#n#") scalar";
    record row(n,t)    = "dim(1:"#n#")"#t;
    record table(m,n,t)= "dim(1:"#m#", 1:"#n#")"#t
{other useful definitions}
obj     = "/usr/dt/matpak/";  {object file directory}
fortran = "FORTRAN";          {source language}
pdp10   = "PDP-10";           {object machine}
```

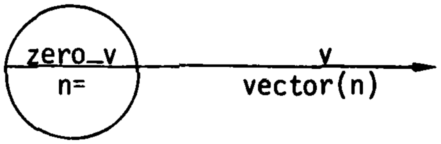
9.1.2 Choice of Nodes

Once the data types are chosen the nodes can be chosen without too much difficulty by inspecting a standard textbook. They are presented in this section both in graphic format and in textual format. Interfaces with the external world, such as module libraries and performance estimates have, of course, been pulled out of a hat.

The nodes can be put roughly into four classes: constructing and disassembling vectors and matrices, simple arithmetic on similarly-sized items, matrix arithmetic, and display of results.

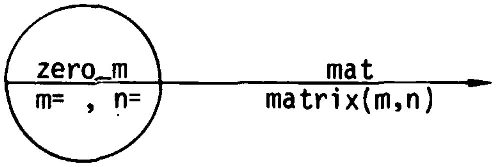
9.1.2.1 Constructing and Disassembling Vectors and Matrices

9.1.2.1.1 Zero Vector. This node exports a vector of length n containing all zeros.



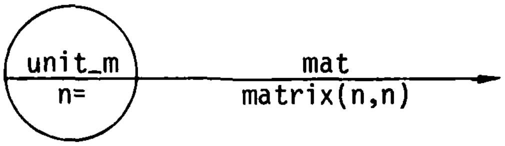
```
node zero_v =

    mod n  = default_n;
    def an = max(1,n);
        unless n = an
        error("n="#n#" is out of range.");
    export vector(an) v;

    begin env(obj#"zerov", fortran, pdp10,
                20+an ms, v:1 out/inv)

    end;
```

9.1.2.1.2 Zero Matrix. This node exports an mxn matrix containing all zeros.



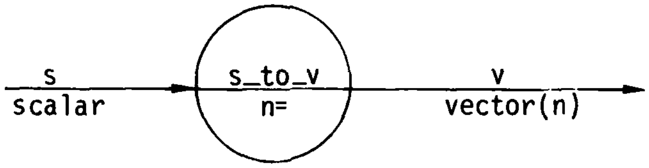
```
node zero_m =

     mod m  = default_m,
         n  = default_n;

     def am = max(1,m);
         unless am=m
         error("m="#m#" is out of range.");
         an = max(1,n);
         unless an=n
         error("n="#n#" is out of range.");
     export matrix(am,an) mat;
     begin env(obj#"zerom", fortran, pdp10,
                25+am*an ms, mat:1 out/inv)
     end;
```

9.1.2.1.3 Unit Matrix. This node exports an nxn unit matrix.



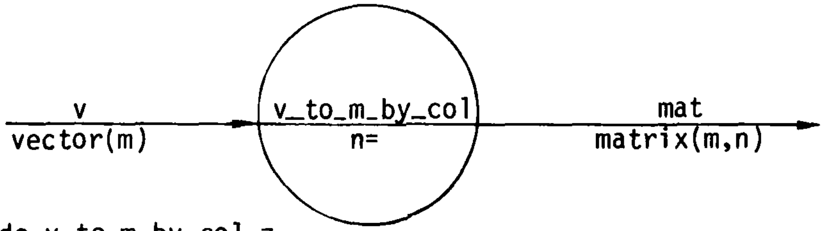
```
node unit_m =

     mod n  = default_n;
     def an = max(1,n);
         unless an=n
         error("n="#n#" is out of range.");
     export matrix (an,an) mat;

     begin env(obj#"unitm", fortran, pdp10,
                30+an**2+2*an ms, mat:1 out/inv)

     end;
```

9.1.2.1.4 Construct Vector from Scalars. This node imports n scalars and then exports a vector of length n consisting of those scalars, in order. The input scalars can be of any 72-bit type.



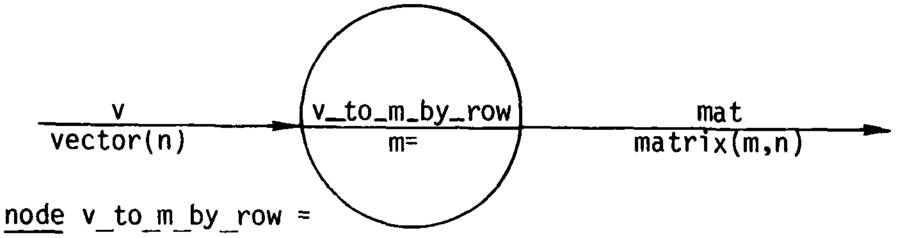
```
node s_to_v =

    import anytype t s;
    mod n  = default_n;
    def an = max(1,min(n,256));
        unless an=n
        error("n="#n#" is out of range.");
        unless length t = 72
        error("size of s="#length t#" is not 72.");
    export row(an,t) v;

    begin env(obj#"stov", fortran, pdp10,
              5 ms, v:an inv/out)

    end;
```

9.1.2.1.5 Construct Matrix from Vectors (by Columns). This node imports n vectors each of length m and then exports an mxn matrix the columns of which are the input vectors, in order. The vectors' elements can be of any 72-bit type.



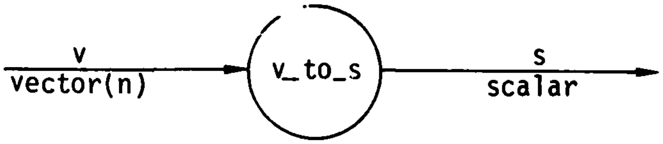
```
node v_to_m_by_col =

    import row(mod m, anytype t) v;
    mod n  = default_n;
    def an = max(1,n);
    unless an=n & m*an<= 4096
    error("n="#n#" out of range.");
    unless length t = 72
    error("size of v[i]="#length t#"is not 72.");
export table(m,an,t) mat;
begin env(obj#"vtombc", fortran, pdp10,
          30+2*m ms, mat: an inv/out)

end;
```

9.1.2.1.6. Construct Matrix from Vectors (by Rows). This node imports m vectors each of length n and then exports an mxn matrix the rows of which are the input vectors, in order. The vectors' elements can be of any 72-bit type.



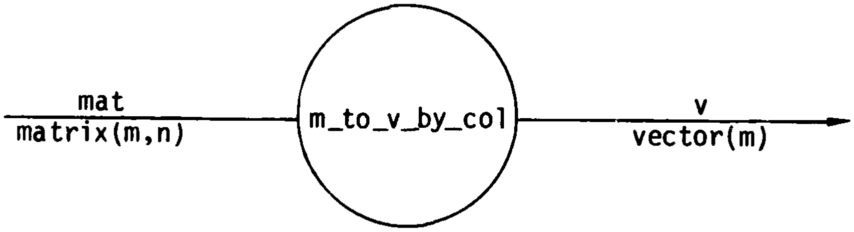
```
node v_to_m_by_row =

    import row (mod n, anytype t)v;
    mod m  = default_m;
    def am = max(1,m);
        unless am=m & am*n<= 4096
        error("m="#m#" out of range.");
        unless length t = 72
        error("size of v[i]="#length t#" is not 72.");
    export table(am,n,) mat;

    begin env(obj#"vtombr", fortran, pdp10,
              30+3*n ms, mat: am inv/out)

    end;
```

9.1.2.1.7 Disassemble Vector into Scalars. This node imports a vector and then exports a series of scalars which were that vector's elements, in order. The scalars may be any 72-bit type.



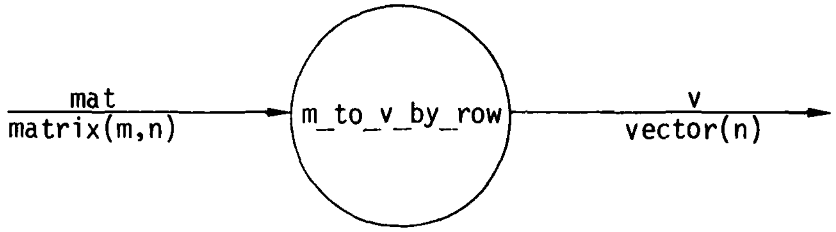
```
node v_to_s =

    import row(mod n, anytype t)v;
    def unless length t = 72
        error ("size of v[i]="#length t#" is not 72.");
    export t s;

    begin env(obj#"vtos", fortran, pdp10,
            10+2*n ms, s: n out/inv)

    end;
```

9.1.2.1.8 Disassemble Matrix into Vectors by Columns. This node imports a matrix and exports a series of vectors consisting of that matrix's columns, in order. The entries of the matrix may be any 72-bit type.



```
node m_to_v_by_col =

    import table (mod m, mod n, anytype t) mat;
    def unless m*n<= 4096
        error("mat's size="#m*n#" is too large.");
        unless length t = 72
        error("size of m[i,j]="#length t#" is not 72.");
    export row(m,t) v;

    begin env(obj#"mtovbc", fortran, pdp10,
            15+2*m*n ms, v: out/inv)

    end;
```

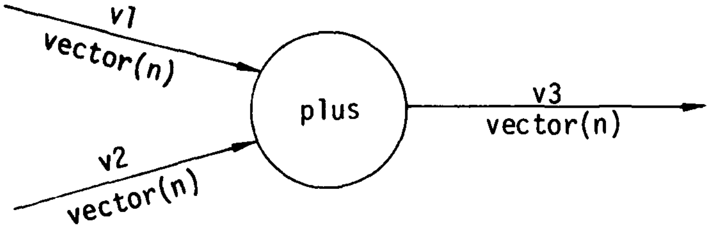
9.1.2.1.9 Disassemble Matrix into Vectors by Rows. This node imports a matrix and exports a series of vectors consisting of that matrix's rows, in order. The entries of the matrix may be any 72-bit type.



```
node m_to_v_by_row =

    import table(mod m, mod n, anytype t) mat;
    def unless m*n<= 4096
        error("mat's size="#m*n#" is too large.");
        unless length t = 72
        error("size of m[i,j]="#length t#" is not 72.");
    export row(n,t) v;

    begin env(obj#"mtovbr", fortran, pdp10,
              15+3*m*n ms, v: m out/inv)

    end;
```

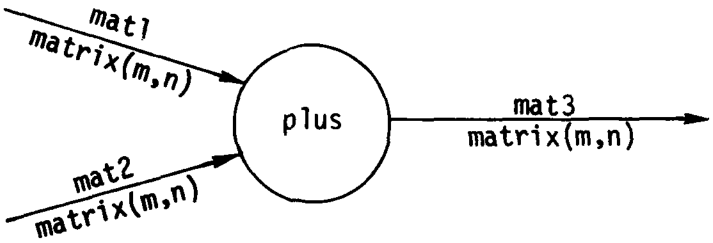
9.1.2.2 Simple Arithmetic on Similarly Sized Items

9.1.2.2.1 Add Vectors. This node imports two vectors and exports their sum.



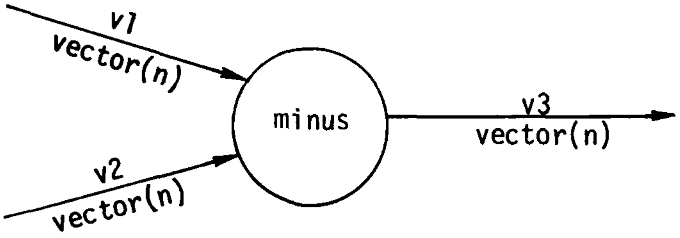
```
node plus =

     import vector (mod n) v1,
            vector (mod np) v2;
     def unless n=np
         error(n#" and "#np#" are incompatible sizes.");
     export vector(n) v3;

     begin env(obj#"plusv", fortran, pdp10,
                20+3*n ms, v3: 1 out/inv)

     end;
```

9.1.2.2.2 Add Matrices. This node imports two matrices and exports their sum.



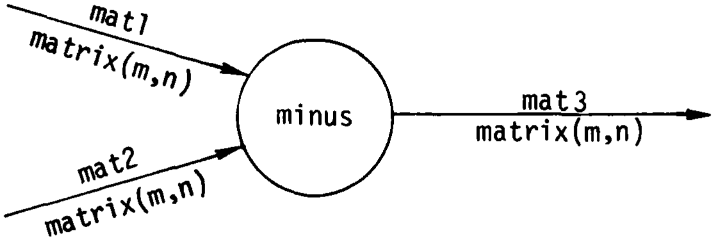
```
node plus =

     import matrix(mod m, mod n) mat1,
            matrix(mod mp, mod np) mat2;
     def    unless m=mp & n=np
            error("("#m#","#n#") and ("#mp#","#np#")"#
                 "are incompatible sizes.");
     export matrix(m,n) mat3;

     begin env(obj#"plusm", fortran, pdp10,
                30+3*m*n ms, mat3: 1 out/inv)

     end;
```

9.1.2.2.3 Subtract Vectors. This node imports two vectors and exports their difference.



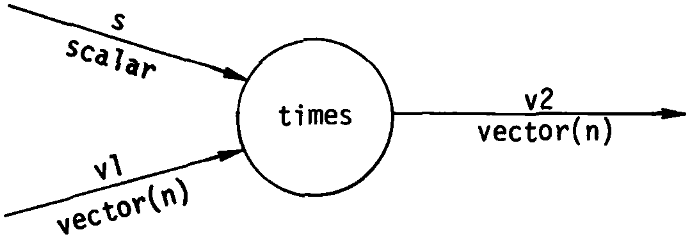
```
node minus =

     import vector (mod n) v1,
            vector (mod np) v2;
     def    unless n=np
            error(n#" and "#np#" are incompatible sizes.");
     export vector(n) v3;

     begin env(obj#"minusv", fortran, pdp10,
                 20+3*n ms, v3: 1 out/inv)

     end;
```

9.1.2.2.4 Subtract Matrices. This node imports two matrices and exports their difference.



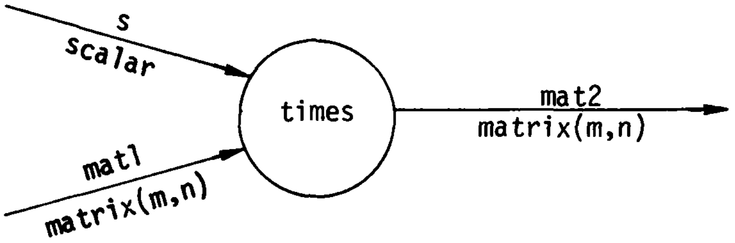
```
node minus =

     import matrix (mod m, mod n) mat1,
            matrix (mod mp, mod np) mat2;
     def    unless m=mp & n=np
            error("("#m#","#n#") and ("#mp#","#np#")"#
                  "are incompatible sizes.");
     export matrix(m,n) mat3;

     begin env(obj#"minusm", fortran, pdp10,
               30+3*m*n ms, mat3: 1 out/inv)

     end;
```

9.1.2.2.5 Scalar Times Vector. This node imports a scalar and a vector and exports their product.



```
node times =

     import scalar s,
            vector(mod n) v1;
     export vector(n) v2;

     begin env(obj#"timesv", fortran, pdp10,
               15+10*n ms, v2: 1 out/inv)

     end;
```

9.1.2.2.6 Scalar Times Matrix. This node imports a scalar and a matrix and exports their product.

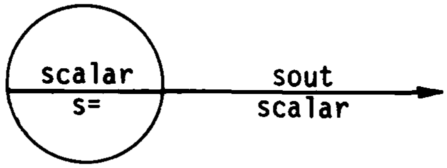


```
node times =

     import scalar s,
            matrix(mod m, mod n) mat1;
     export matrix(m,n) mat2;

     begin env(obj#"timesn", fortran, pdp10,
               35+10*m*n ms, mat2: 1 out/inv)

     end;
```

9.1.2.2.7 Scalar Operations. It is assumed that there are plus, minus, times, divided by, etc., nodes defined on single scalars with the obvious semantics. These definitions can be provided by a call on a previously existing library "scalib." This would appear in the current library as the library-inclusion

```
lib scalib;
```

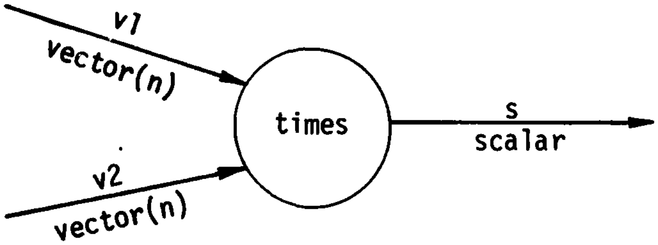
Perhaps the most important node in scalib from the current point of view is the node which exports a single scalar. For convenience this node definition, which actually appears within "scalib," is repeated here. This node exports the scalar that is represented by the string which passed to its modifier.



```
node scalar =

     mod s = "0.0"; {default output is zero}
     export scalar sout;
     begin env("/usr/dt/scalib/scalar", fortran, pdp10,
               50 ms, sout: 1 out/inv)

     end;
```

9.1.2.3 Matrix Arithmetic

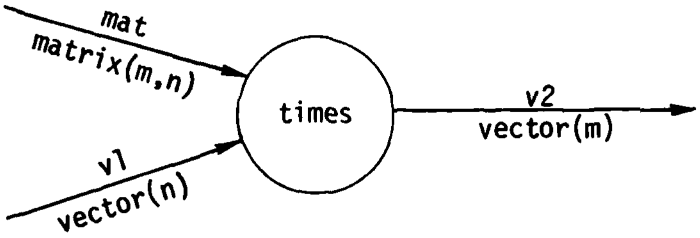
9.1.2.3.1 Vector Innerproduct. *This* node imports two vectors and then exports their innerproduct.



```
node times =

     import vector (mod n) v1,
            vector (mod np) v2;
     def    unless n=np
            error("size of v1="#n#" and size of v2="#np#
                  "are incompatible");
     export scalar s;
```

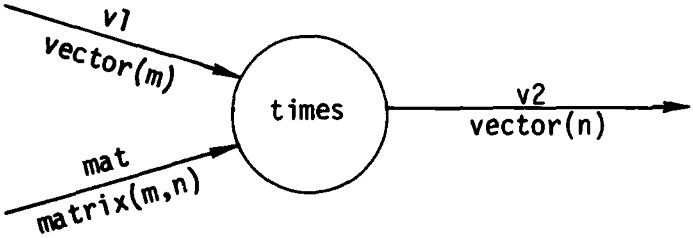
```
        begin env(obj#"inner", fortran, pdp10,
                20+6*n ms, s: 1 out/inv)

        end;
```

9.1.2.3.2 Matrix Times Column Vector. This node imports a matrix and vector and exports their product.



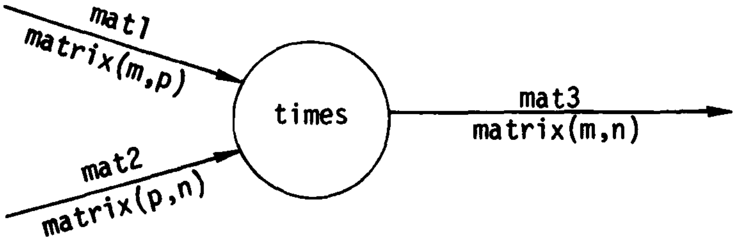
```
        node times =

            import matrix(mod m, mod n) mat,
                   vector(mod np) v1;
            def    unless n=np
                   error("size of mat=("#m#","#n#") is "#
                         "incompatible with size of v1="#np);
            export vector(m) v2;

            begin env(obj#("timemv"), fortran, pdp10,
                      35+m*(20+6*n) ms, v2: 1 out/inv)

            end;
```

9.1.2.3.3 Row Vector Times Matrix. This node imports a vector and a matrix and exports their product.



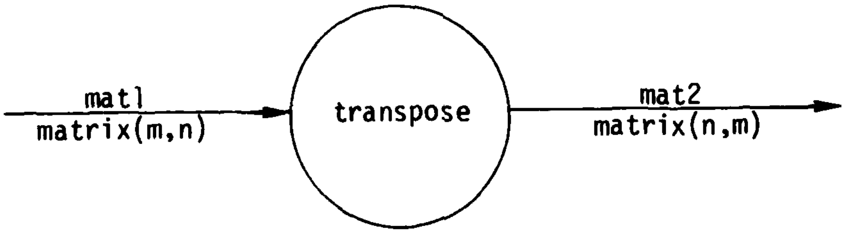
```
node times =

     import vector(mod m) v1,
            matrix(mod mp, mod n) mat;
     def    unless m=mp
            error("size of v1="#m#" and of "#
                  "mat=("#mp#","#n#") are incompatible);
     export vector(n) v2;
     begin env(obj#"timevm", fortran, pdp10,
                35+n*(20+6*m)ms, v2: 1 out/inv)
     end;
```

9.1.2.3.4 Matrix Times Matrix. This node imports two matrices and exports their matrix product.



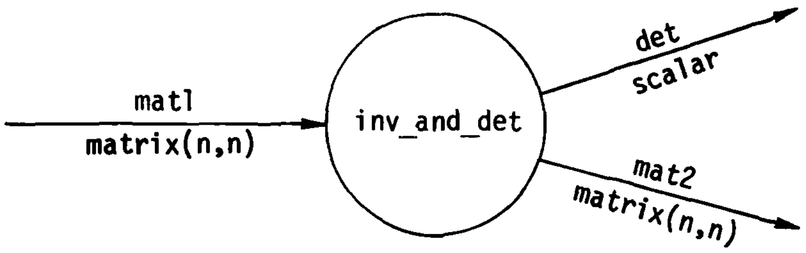
```
node times =

     import matrix(mod m, mod p) mat1,
            matrix(mod pp, mod n) mat2;
     def    unless p=pp
            error("size of mat1 = ("#m#","#p#")"
                  #"and of mat2 = ("#pp#","#n#")"
                  #" are incompatible.");
     export matrix(m,n) mat3;

     begin env(obj#"timemm", fortran, pdp10,
               100+n*(35+m*(20+6*p))ms,
               mat3: 1 out/inv)

     end;
```

9.1.2.3.5 Transpose Matrix. This node imports a matrix and exports its transpose.



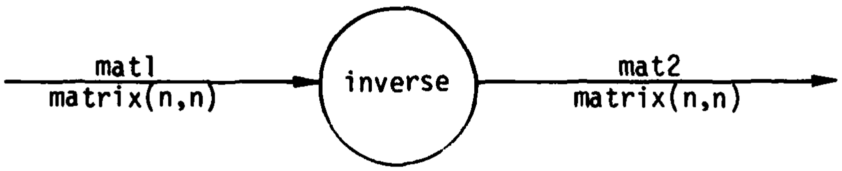
```
node transpose =

     import table(mod m, mod n, anytype t) mat1;
     def    unless length t = 72
            error("size of m[i,j]="#length t#" is not 72.");
     export table(n, m, t) mat2;

     begin env(obj#"trans", fortran, pdp10,
               35+2*m*n ms, mat2: 1 out/inv)

     end;
```

9.1.2.3.6 Inverse and Determinant of Matrix. This node imports a square matrix and exports its determinant and inverse.



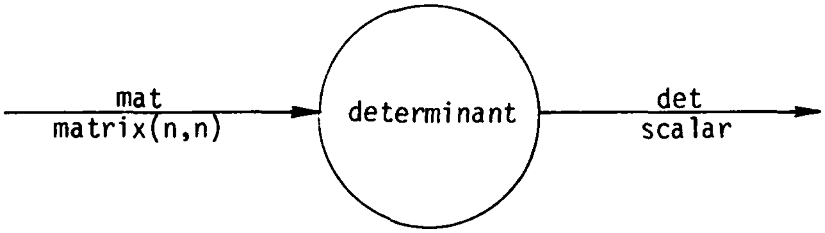
```
node inv_and_det =

    import matrix(mod n, mod np) mat1;
    def    unless n=np
           error("cannot take inverse of nonsquare "#
               "matrix of size = ("#n#","#np#").");
    export scalar det,
           matrix(n,n) mat2;

    begin env(obj#"invdet", fortran, pdp10,
             250+6*n**3 ms,
             det: 1 out/inv,
             mat2: 1 out/inv)

    end
```

9.1.2.3.7 Inverse of Matrix. This node imports a square matrix and exports its inverse. It is defined in terms of (9.1.2.3.6).



```
node inverse =

    import matrix(mod n, mod np) matl;
    export matrix(n,n) mat2;

    begin
       inv_and_det(port matl) --> *, port mat2

    end;
```

9.1.2.3.8 Determinant of Matrix. This node imports a square matrix and exports its determinant. It is defined in terms of paragraph 9.1.2.3.6.



```
node determinant =

    import matrix(mod n, mod np) mat;
    export scalar det;

    begin
       inv_and_det(port mat) --> port det, *

    end;
```

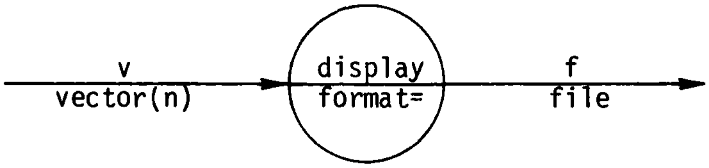
9.1.2.4 Display of Results

Once a useful vector or matrix has been calculated, it should be displayed. To be useful, display requires a number of options which are closely related to the underlying machine. Because the matrix library is an example of something well understood, it was not thought advisable to design display nodes in detail; rather, they will use notions that are purposely left undefined here.

A display node will take as input a vector or matrix and produce an output on a file port; "file" is purposely left undefined but can be thought of a readable set of characters that displays the imported record. Control of this display is specified by a "format", which is some character string; a format can be thought of as a Fortran format specification.

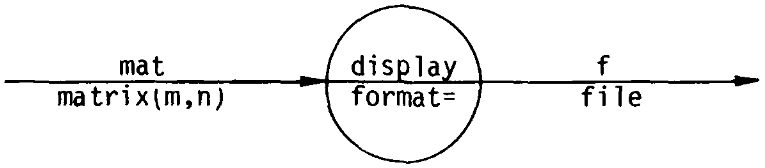
Presumably the scalar library (see paragraph 9.1.2.2.7) contained an appropriate definition for the "file" record type so it need not be repeated in this library.

9.1.2.4.1 Display Vector. This node displays its import vector according to the format specified.



```
node display =

     import vector(mod n) v;
     mod    format;
     def    unless n<=24
            warn("vector("#n#") will not "#
                 "fit on screen.");
     export file f;

     begin env(obj#"dispv", fortran, pdp10,
               200+100*n ms, f: 1 out/inv)

     end;
```

9.1.2.4.2 Display Matrix. This node displays its import matrix according to the format specified.



```
node display =

     import matrix(mod m, mod n) mat;
     mod    format;
     def    unless m*n<=24
            warn ("matrix("#m#","#n#") will not"
                  # "fit on screen.");
     export file f;

     begin env(obj#"dispm", fortran, pdp10,
               200+100*m*n ms, f: 1 out/inv

     end;
```

9.2 A SAMPLE PROGRAM USING THE MATRIX LIBRARY

This section demonstrates how the library given in Section 9.1 can be used. The example arises in minimizing time delay in message-switched communication networks. The problem is to do many calculations of

$$g = f - hP_q \nabla T$$

where f is a flow vector, P_q is a projection operator matrix, T is a function of f which is to be minimized, h is a step size to be determined by the experimenter as the "experiment" is running, and g is the output flow vector.

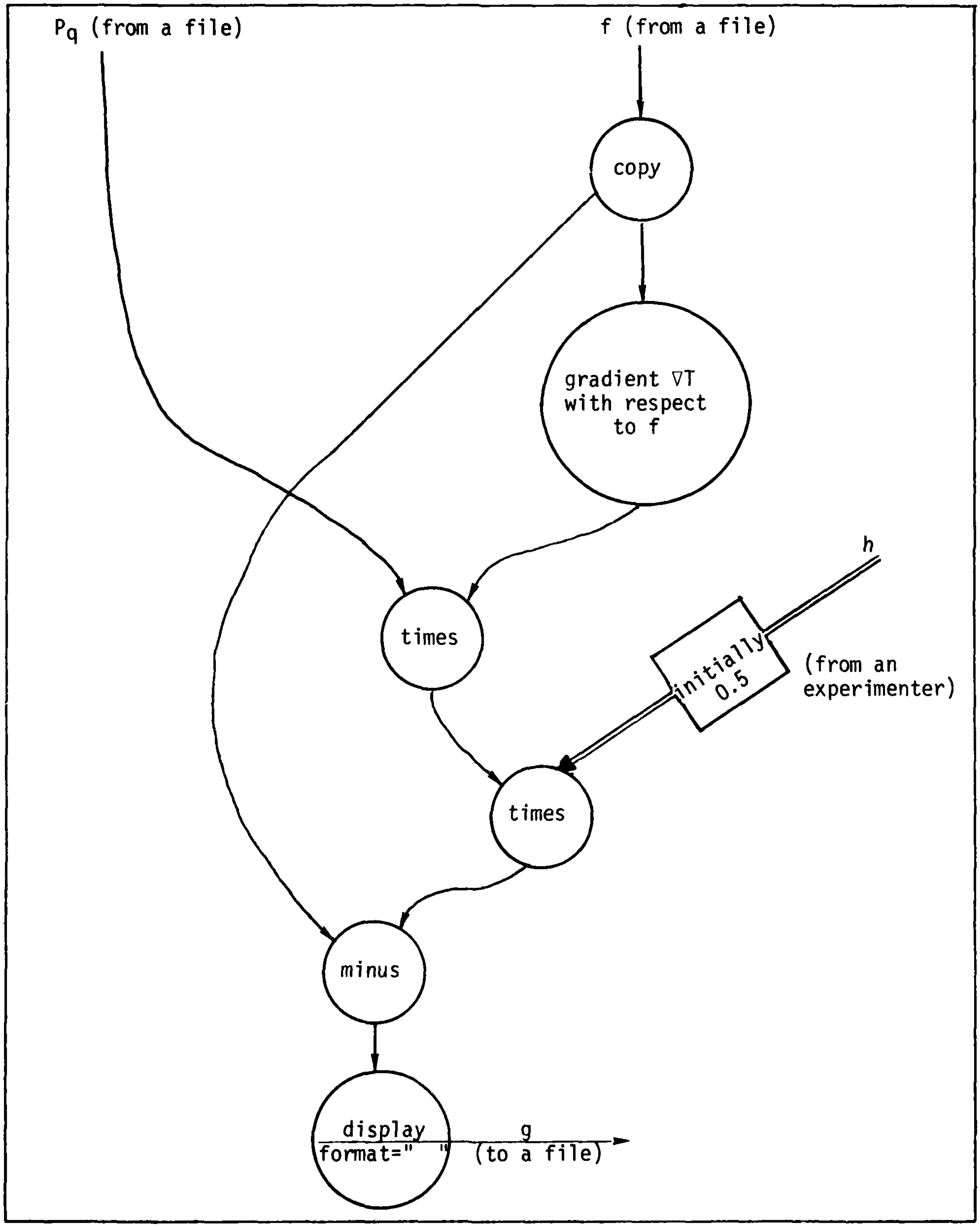
A network that performs this calculation is displayed in Figure A-1. This network reads successive pairs of P_q and f, polls the experimenter for an appropriate h, and displays the results.

Because the "gradient ∇T with respect to f" node is special for this experiment and is not in the matrix library, it must be written either in LINGO or in an algorithmic language as a primitive node. Because it is written only for this particular network, it need not be as qeneral as the nodes in the matrix library. Its specification might run as follows:

```
node grad_t_f =
    import vector (150) f;
    export vector (150) del_t;
    begin env ("/u/dt/gradtf",
                fortran,
                pdp10,
                1000 ms,
                del_t: 1 out/inv)
    end;
```

Once the gradient node has been coded, the network in Figure A-1 may be expressed in a character form as follows:

FIGURE A-1 NETWORK FOR $g = f - HP_q\nabla T$

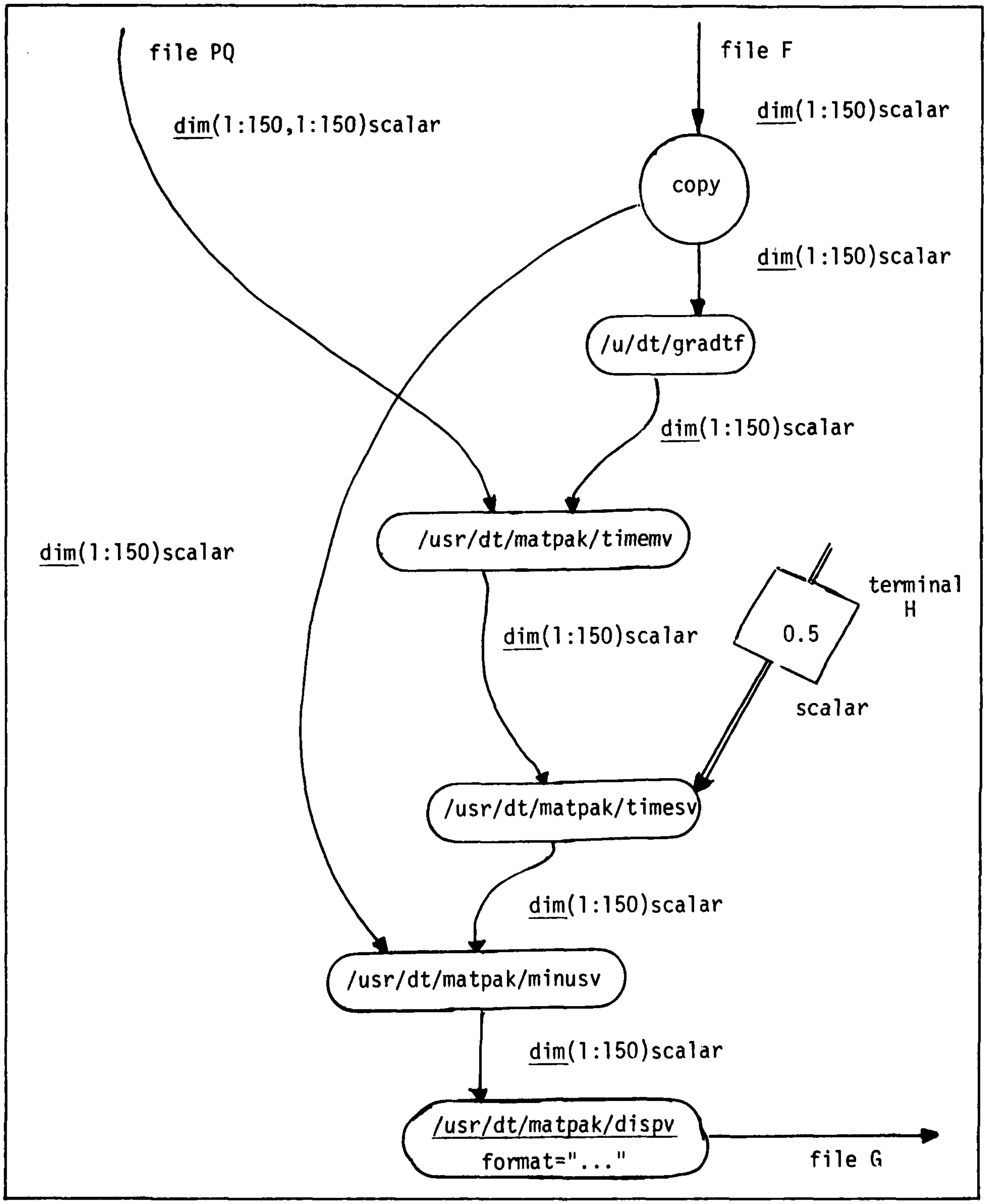


```
def lib matpak; {include matrix library}
    node grad_t_f =
        {mentally insert the definition for
         "grad_t_f" here};
import matrix (150, 150) pq,
    vector (150) f;
export file g;
begin
    copy port f --> f1, f2;
    times (port pq, grad_t_f(f2))--> a;
    times (<scalar(/s = "0.5"), port h>, a)--> b;
    display (minus(f1,b)/format="...")--> port g
end
```

Notice that "times" is generic; when first used in the program it identifies the matrix times column vector routine; when used second it identifies the vector times scalar routine. Both identifications are done at link time, before the program gets executed.

The linked version of the sample program is shown in Figure A-2, which shows the object module for each node and the record types to be found on each link. This figure is a pictorial representation of the tables that the network linker will output to the system that executes LINGO networks.

FIGURE A-2 LINKED NETWORK FOR $g = f - hP_q\nabla T$



10.0 CONCLUSIONS

This report has concentrated on defining a language that allows a casual user to compose a data driven network of library programs which perform useful work. The level of composition encourages simple constructions rather than extensive conditional and looping constructs to achieve the desired result. Therefore the user needs only to understand the area in which computation will be done and LINGO's simple rules which connect operational nodes together to perform computations.

The majority of LINGO, and therefore the bulk of this document, concentrates on LINGO facilities for the applications library designer. These include methods of specifying the performance of a node in terms of the user's requirements so that the cost of executing a network can be predicted in advance; methods for checking proper interconnection of application nodes; and methods for predefining entities for users to make their use of LINGO system easier.

In doing the design of the Network Language LINGO, several ideas have been discarded because they added yet another feature for the applications library designer in an area that was already the largest part of the language. Without doing the design of several libraries and evaluating the current complement of facilities, it is difficult to know whether these additional features are needed.

It is important to note that this report is a preliminary version and that undoubtedly LINGO will be changed as a result of experience in node library design. For example, one desirable, though perhaps impossible, change would be to simplify the notions of formal- and actual-record types without taking away any power or convenience.

Probably the most important extension considered was "alternative records" so that more than one type of record could traverse a given link. We have also considered methods for an import to consume more than one record at a time during one invocation. This can be done now by insertion of a node which combines together the desired amount of data and exports it. It is not clear that more than this is really needed. For every simplification and extension, care must be taken that the original goals are met and that deadlock has not crept in the back door.

After several application libraries are designed and some experience with LINGO has been attained, changes should be integrated delicately and inconsistencies repaired beautifully.